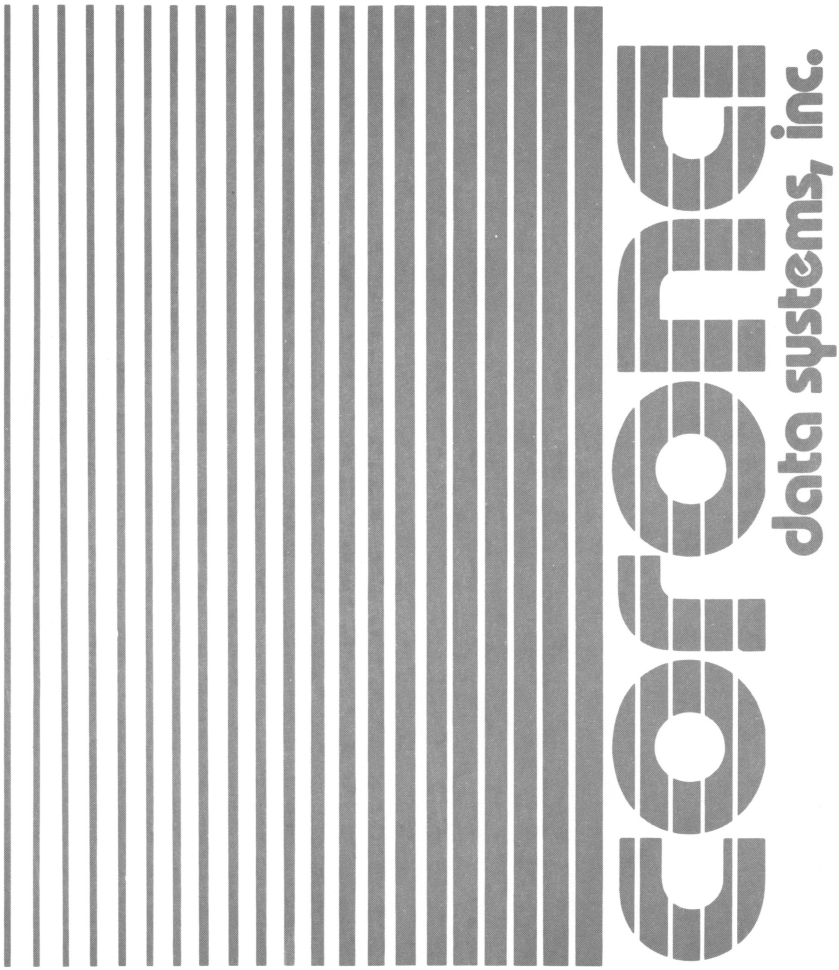
GWBASIC 2.0



corona
data systems, inc.

GWBASIC 2.0

Part No. 700653
Rev. A

CONTENTS

FIGURES

TABLES

PREFACE

The information below provides details of the hardware and software requirements for running GWBASIC 2.0, together with a list of the features that are new in this release.

Existing features of GWBASIC that have changed slightly for 2.0 are listed in the README.DOC file on the GWBASIC 2.0 diskette (note that this is an ASCII file, not a word processing file). To see the contents of this file, use the DOS commands TYPE (to display the file on the screen) or PRINT (to list the file on a printer).

HARDWARE AND SOFTWARE REQUIREMENTS

Version 2.0 of GWBASIC is designed to run on the following systems:

- 400-line Desktop and Portable Computers (PC-400 and PPC-400)
- 325-line Desktop and Portable Computers
- MEGA PC

The hardware configuration required for this version of GWBASIC is as follows for all systems:

- At least 256 Kbytes of memory
- SYSTEM ROM version 3.10 or later

Operating system software required to run GWBASIC 2.0 is:

- DOS 2.0 or later releases

GWBASIC version 2.02 and later releases are compatible with BASIC 3.0 used on the IBM PC.

NEW FEATURES IN THIS RELEASE

Several new features have been added to GWBASIC since the previous release (1.xx). These features are:

- Redirection of standard input (INPUT and LINE INPUT statements) and output (PRINT statement)
- Support of character devices, allowing user-installed device drivers to interface with GWBASIC (IOCTL statement and IOCTL$ function - see also Section 3)
- Improved disk I/O facilities for handling larger files (GET and PUT statements (files))
- SHELL statement, allowing DOS commands or "child" (i.e., called) processes to be executed without having to leave GWBASIC
- Tree-structured directory management, to take full advantage of DOS 2.0 disk file organization (MKDIR, CHDIR and RMDIR statements and use of pathnames in file specifications)

New and improved graphics features:

- Definition of viewports within current screen (VIEW and VIEW PRINT statements)
- Redefinition of screen or viewport coordinates (WINDOW statement)
- PMAP function, allowing "world coordinates" created by the WINDOW statement to be mapped to physical screen coordinates and vice versa
- Form of POINT function to return value of current graphics coordinates (physical or logical)
- Line clipping instead of wraparound where a line continues outside the screen or viewport boundary (all graphics statements)
- Line styling, allowing lines to be drawn in dotted and/or dashed patterns (LINE statement)
- Paint tiling, allowing a figure to be painted with a pattern (PAINT statement)

- Turn angle (TA) and paint (P) commands for use when drawing a figure (DRAW statement)

Other features:

- Screen editor enhancements, including text window support
- TIMER function, returning number of seconds elapsed since midnight or system reset
- Use of TIMER function to seed random number generator (RANDOMIZE statement)
- Music and timer event trapping (ON PLAY(n)... and ON TIMER(n)... statements)
- Easier octave changing for music (PLAY statement)
- User-defined key sequences (KEY statement) and trapping (ON KEY(n)... statement)
- ERDEV and ERDEV$ functions, returning device error code and device driver name when an error occurs
- Double-precision option for standard math functions (/D switch of GWBASIC command)
- Improved control of memory allocation for assembly language routines (/M switch of GWBASIC command)
- Deletion from specified line number to end of program (DELETE statement)
- OPTION BASE allowed in chained programs (OPTION BASE statement)
- DATA statements RESTOREd before chained program is run (CHAIN statement)

Section 1
INTRODUCTION

This is your reference manual for the GWBASIC 2.0 Interpreter. The following sections present a comprehensive overview of this program and how to use it. First you will learn how to bring up GWBASIC and to use it to run existing programs or to write your own programs. Following that is specific information on files, graphics, characters, numbers, and communications. The main reference section contains an alphabetical listing of the commands, statements, functions, and variables that can be used in GWBASIC. In the appendices is additional, more technical, information on files and graphics, as well as a list of ASCII codes, keyboard scan codes, reserved words, trigonometric functions, and error messages. Other appendices provide programming hints, a list of recommended reading, and a summary of GWBASIC commands.

NOTE: This manual is not intended to be a tutorial on BASIC programming. It assumes that you have a working knowledge of BASIC. If you are new to BASIC and need to find out more about it, read one of the BASIC tutorials listed in Appendix G.

GWBASIC 2.0 runs under DOS version 2.0 or later. Some sections of this manual have different instructions according to what type of system you have. If you have a 325-line desktop or portable PC, you should follow the instructions for 325-line users. If you have a 400-line desktop or portable PC (PC-400 or PPC-400), or a MEGA PC, you should follow the instructions for 400-line users.

Note that information on sound and on color monitors applies to 325-line, PC-400 and PPC-400 systems only.

Section 2
GETTING GWBASIC STARTED

BEGINNING AT DOS

The procedural steps given throughout this section assume that you have two floppy drives (A: and B:). If you have a single-drive system, you may have to keep switching diskettes to perform some steps (for instructions on how to invoke procedures with a single drive, read Appendix A in your DOS manual). If you have a system with a hard drive, booting as well as other procedures may be different (again, refer to the DOS manual for detailed information).

Perform the following steps to start GWBASIC:

1. Insert the DOS disk in your boot drive.

2. Switch the computer on.

3. At the A> prompt type:

```
GWBASIC <Return>
```

You will see the version number, number of free bytes, and the GWBASIC prompt "Ok".

For more information about the options that you can specify when invoking GWBASIC, see the GWBASIC command specification in Section 7.

USING BASIC OR BASICA PACKAGES

The following instructions will explain how to use BASIC or BASICA packages on your computer:

1. Read the installation information provided with the package to determine whether it runs under BASIC or BASICA. If this information cannot be found, check the directory of the product disk and TYPE the batch file that loads the program. This will specify whether BASIC or BASICA is used. For example, if the batch file is AUTOEXEC.BAT, the following would be entered at the A> prompt

after booting with DOS and inserting the product disk in the boot drive:

```
A>TYPE A:AUTOEXEC.BAT <RETURN>
```

NOTE: If you have two diskette drives, put DOS in drive A and the product in drive B. Then specify B: instead of A: to call up the batch file on drive B.

The following should appear on the screen:

```
BASIC filename
```

or

```
BASICA filename
```

Other commands may also be listed in this batch file, and there may be parameters listed after **filename**.

2. To create a self-booting backup product disk, make sure the disk is not copy-protected (if it is, read the paragraph below). Then format a disk with the system on it using the FORMAT /S command. Finally, perform a COPY command to copy GWBASIC.EXE along with either BASIC.COM or BASICA.COM onto the disk (whether you copy BASIC.COM or BASICA.COM depends on your specific product disk). For information on the FORMAT and COPY commands, see "FORMAT" and "COPY" in the DOS manual.

   If the product disk is copy-protected, copy GWBASIC.EXE onto it, along with either BASIC.COM or BASICA.COM. This, disk can then be used after booting with DOS.

3. The following steps show what to do if the product and GWBASIC do not fit on the same disk.

   It is assumed that you have a system with two diskette drives. If you have a single drive, you must keep switching diskettes. Users with single-drive systems should refer to the DOS manual for insructions on how to invoke procedures with a single drive.

   a. Boot the computer with the DOS disk in drive A.

   b. Insert the product disk in drive B and TYPE the batch file that loads it.

c. At the A> prompt, run the files, if any, preceding the BASIC or BASICA command.

d. Load GWBASIC from the DOS disk and insert the product disk in drive A. Type the following:

```
RUN "filename" <Return>
```

If the product does not have an AUTOEXEC.BAT file, only load GWBASIC from the DOS disk, then insert the product disk and at the **Ok** prompt type:

```
RUN "filename"
```

If you have a hard drive, you may copy the product disk to it as long as the product disk is not copy-protected.

NOTE: If you have a 325-line system, you cannot boot from the hard drive. You can boot from the hard drive if you have a PC-400, PPC-400, or a MEGA PC. For more information on booting from the hard drive, refer to the DOS manual.

WRITING YOUR OWN PROGRAMS

When GWBASIC is invoked, it displays the prompt:

```
Ok
```

This means that GWBASIC is at the command level ready to accept commands. At this point, GWBASIC may be used in either of two modes: the direct mode or the indirect mode.

Direct Mode

In this mode, statements and commands are not preceded by line numbers, but are executed as they are entered, for example:

```
Ok
A = 5 + 6
Ok
PRINT A
 11
Ok
```

Arithmetic and logical operation results are displayed immediately and stored for later use, but instructions are lost after execution. This mode can be useful for debugging and for using BASIC as a calculator for quick computations that do not require a complete program.

Indirect Mode

This mode is used for entering programs. Program lines are preceded by line numbers and are stored in memory. The "RUN" command executes the program stored in memory. Here is the above example in indirect mode:

```
10 A = 5 + 6
20 PRINT A
RUN
 11
Ok
```


Line Format

Program lines in a BASIC program have the following format:

```
nnnnn BASIC-statement[:BASIC-statement...]
```

Square brackets indicate optional input.

More than one BASIC statement can be placed on a line, but each statement on a line must be separated by a colon.

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

Line numbers indicate the order which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. A period may be used in EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

It is possible to extend a logical line over more than one physical line by using the Ctrl-Enter keys. Ctrl-Enter lets you continue typing a logical line on the next physical line without entering a Return.

EDITING

GWBASIC's editor can save you a sizable amount of time during the development of your programs.

Any line of text typed while GWBASIC is in direct mode will be processed by the editor. GWBASIC is in direct mode after the Ok prompt and until a RUN command is invoked.

NOTE: GWBASIC's editor will convert lower-case entries to upper-case, except for remarks, DATA statements, and strings enclosed in quotation marks.

If there are more than 255 characters in one line, the extra ones will be truncated when Return is pressed. They will appear on the screen but will not be processed.

Program Editor Keys

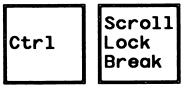
Home	Moves the cursor to the upper left corner of the screen.
Ctrl + Home	Clears the screen and positions the cursor in the upper left corner of the screen.
↑	Moves the cursor up one line.
↓	Moves the cursor down one line.
←	Moves the cursor one position left. When the cursor is advanced beyond the left of the screen, it will be moved to the right side of the screen on the preceding line.
→	Moves the cursor one position right. When the cursor is advanced beyond the right of the screen, it will be moved to the left side of the screen on the next line down.

Ctrl →	Moves the cursor one word to the right.
Ctrl ←	Moves the cursor one word to the left.
End	Moves the cursor to the end of the logical line. Characters typed from this point are appended to the line.
Ctrl End	Erases the cursor position to the end of the logical line.
Ins	Toggles Insert mode on or off. Insert mode is indicated by the blinking cursor covering the lower half of the character position. In graphics modes, the normal cursor covers the whole character position. When in Insert mode, characters following the cursor are moved to the right as characters are inserted at the current cursor position. The characters that advance off the right side of the screen are inserted from the left on the following lines. When out of Insert mode, characters typed will replace existing characters on the line.
Tab	When out of Insert mode, this key moves the cursor over characters until the next tab stop is reached. Tab stops occur every eight character positions. When in Insert mode, characters following the cursor are moved to the right, causing blank spaces to be inserted from the current cursor position to the next tab stop. The characters that advance off the right side of the screen are inserted from the left on the following lines.

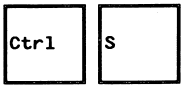
Deletes the character at the cursor position. All characters to the right of the one deleted are moved one position left. If a logical line extends beyond a physical line, the character in the first column of each subsequent line is moved up to the end of the preceding line.

Causes the last character typed to be deleted, or deletes the character to the left of the cursor. All characters to the right of the cursor are moved one position left. Subsequent characters and lines within the logical line are moved up as with the Del key.

Causes the entire logical line to be erased. May be typed anywhere on the line.

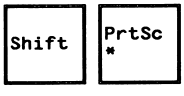


Returns to Direct mode without saving any changes that were made to the current line being edited.

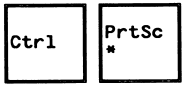


Freezes the screen; useful when listing a program on the screen that is scrolling too fast for you to read it. Pressing Ctrl-S a second time causes the listing to continue.

Printing Keys



Causes the entire contents of the screen to be printed out. Pressing Shift-PrtSc a second time stops this printing.



Causes anything typed after these keys are pressed to be printed out when Return is pressed. If you press Ctrl-PrtSc a second time, this screen echoing will stop.

Ctrl Key

The Ctrl key performs various additional functions if used in combination with certain alphabetic keys. These functions are listed in Table 2-1, together with other keys that produce the same effect.

Table 2-1

CTRL KEY FUNCTIONS

Ctrl +	Equivalent	Action
B	Ctrl- ←	Moves cursor back to previous word
C	Ctrl-Break	Interrupts program execution and returns to direct mode
E	Ctrl-End	Erases from cursor to end of current line
F	Ctrl- →	Moves cursor forward to next word
G	-	Sounds the speaker
H	Back Space	Deletes character to left of cursor
I	Tab	Moves cursor eight positions to the right
J	Ctrl-Enter	Inserts blank line after current line
K	Home	Moves cursor to upper left corner of screen
L	Ctrl-Home	Clears screen and homes cursor
M	Enter	Enters text typed on current line into memory
N	End	Moves cursor to end of current line
R	Ins	Toggles insert mode on and off
S	Ctrl-Num Lock	Toggles suspension of program execution on and off

Table 2-1 (Cont.)

Ctrl +	Equivalent	Action
T	KEY ON/OFF	Toggles display/hide of function key values
U	Esc	Deletes current line
W	-	Deletes word at current cursor position
Z	Ctrl-Pg Dn	Clears from cursor to end of screen

Adding New Lines

Enter a valid line number followed by at least one character. When you press Return this line will be saved in memory. Valid line numbers are 0 to 65529. If a line already exists with the same line number, the old line will be replaced by the new one. If you run out of memory while entering text, the following error message will occur:

```
Out of memory
```

That line of text will not be added.

Replacing Existing Lines

Enter the number of the line to be replaced followed by the desired replacement text. When Return is pressed, the new line will replace the old.

Deleting Lines

Type the line number of the line to be erased, and then press Return. The line will be erased from the program. Esc will erase a line on the screen, but that line will continue to exist in the program.

Duplicating Lines

Move the cursor to the number of the line you wish to duplicate, type over the old number with a new number, and then press Return. Both old and new lines will be included in the program.

Altering Lines on the Screen

Use the LIST and EDIT commands (described in Section 7) to display any lines not on the screen.

The cursor movement keys can be used to position the cursor anywhere on the screen. Use any method described previously to manipulate text, and then press Return. The use of Return will enter all changes for that logical line (i.e., up to 255 characters), no matter how many physical lines on the screen are involved and no matter where the cursor is located in the line.

NOTE: Any changes made are only made in memory. To save changes permanently, see below under "Saving Programs".

Deleting a Program

To clear memory before entering a new program, use the NEW command.

To delete a program that has already been saved (see "Saving Programs" below), use the KILL command.

Both of these commands are described in Section 7.

FORMAT ERRORS

When a format error is encountered during program execution, BASIC automatically enters edit mode and displays the line that caused the error. You then make your correction and press Return.

NOTE: Storing the line back in the program causes all variables to be lost. To examine the contents of a variable before making the change, type Ctrl-Break to return to direct mode. The variables would be saved since no program line was changed.

SAVING PROGRAMS

When you write a program under GWBASIC, the program will be lost when you exit GWBASIC unless you save (store) the program first. You can only save a program written in indirect mode. To save a program, execute the SAVE command, which writes the program to a disk file, adding the extension ".BAS" to the filename that you choose, unless you specify a different extension. You can then use the RUN statement to execute any program that has been saved in this way.

EXITING GWBASIC

To exit GWBASIC and return to DOS, type the following at the Ok prompt:

```
SYSTEM <Return>
```

Typing Ctrl-Break or Ctrl-C will not return you to DOS.

Section 3
HANDLING FILES AND DEVICES

The commands and statements used in program files are described in brief in this section, which also includes further information on files, devices, user-installed device drivers, redirection of input and output, and tree-structured directories. Refer to Appendix A for information on random and sequential files.

For information on hardware options and installations, switch settings, connecting system components, diskette and hard drive care and handling, and general information for using the system, refer to the User's Guide. The guide also contains a glossary of commonly used computer terms.

Refer to the DOS manual for information on system start-up, important diskette and hard drive instructions, command format, wildcard characters, device filenames, the system keyboard, instructions for users with single-drive systems, copying and backing up files on entire disks, and DOS commands.

FILE NAMES

Whenever a filename is required in a disk command or statement, you must use a filename that conforms to the naming conventions described in the DOS manual. DOS will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE, or LOAD command.

PROGRAM FILE COMMANDS

Following is a review of the commands and statements used in program file manipulation. For detailed information on GWBASIC commands, refer to Section 7.

SAVE filespec[{,A | ,P}]

Writes the program currently in memory to disk in a compressed binary format. If the A parameter is used, the program will be written as a

series of ASCII characters. Compressed binary format takes up less disk space, but some GWBASIC commands such as MERGE require a program file to be in ASCII format. If P is used, the file is read-protected (see "Protected Files" below).

LOAD filespec[,R]

Loads the program from disk into memory. If the R parameter is used, the program will be run immediately. LOAD will delete any current program from memory and close all open files. If the R parameter is used, open data files are kept open, and programs may be chained or loaded in sections and may access the same data files. (LOAD **filename**,R and RUN **filename**,R are the same.)

RUN filespec[,R]

Loads the program from disk into memory and runs it. RUN will delete the current contents of memory and close all files. If the R parameter is used, open data files are kept open. (RUN **filespec**,R and LOAD **filespec**,R are the same.)

MERGE filespec

Loads the program from disk into memory without deleting the current contents of memory. The program line numbers on disk merge with the line numbers in memory. If two lines have the same number, the line from the disk program replaces the one in memory.

CHAIN [MERGE]filespec[,[line][,ALL][DELETE range]]

Passes control to the specified program, starting it at line number **line** if given. Some or all of the current variables can be passed to the new program, and an overlay can be brought in or deleted.

KILL filespec

Deletes the file from the disk. **filename** may be a program file, or a sequential or random access data file.

NAME old filespec AS new filespec

Changes the name of a disk file. May be used with program files, random files, or sequential files.

NOTE: Wildcard characters, allowing you to reference more than one file in a single command, can be used with the FILE, KILL, and

NAME commands. Further details about wildcard characters are given in the DOS manual. Use these characters with great care - an incorrect use of them with KILL, for example, could result in the deletion of many more files than you intended.

PROTECTED FILES

You can save a program in an encoded binary format with the Protect parameter (P). This parameter is used with the SAVE command as follows:

```
SAVE filespec,P
```

A program saved in this manner cannot be listed or edited.

FILE AND DEVICE INFORMATION

A file contains information such as a GWBASIC program or data used by a program. In order to use the information, you must specify where it is to be found, using a file specification (filespec). A filespec is a string expression (and must therefore be enclosed in quotes whenever it is specified to GWBASIC) and has the following form:

```
[device:][[\directory][\directory...]]filename
```

Directories are explained below under "Tree-Structured Directories". A filespec that includes directory names, telling the system which route to take to find a disk file, is called a **path**.

The device name tells the system which device the file is on. The name consists of up to four characters followed by a mandatory colon. (Note that in DOS, however, the colon is not mandatory.) Table 3-1 shows a list of device names with their references and indicates whether they can be used for input or output. If you omit the device name from the filespec, the default is the drive that was the DOS default before GWBASIC was invoked.

The filename is the name of your file. It must conform to the naming conventions described in the DOS manual. The only difference is the legal characters that can be used in the name and extension. Only the following characters are allowed:

A through Z 0 through 9 @ # $ % & ! -

Table 3-1

DEVICE INFORMATION

Code	Name	Use
KYBD:	Keyboard	Input only
SCRN:	Screen	Output only
LPT1:	First printer	Output or random
LPT2:	Second printer	Output or random
COM1:	First asynchronous communications adapter (on-board)	Input and output
COM2: COM3: COM4:	Add-on asynchronous communications adapters (if used)	Input and output
A:	First diskette drive	Input, output, and random
B:	Second diskette drive	Input, output, and random
C:	Hard drive (see note)	Input, output, and random

NOTE: On the MEGA PC, the first partition on the hard disk is C:, and the remainder are lettered from D: onwards. The tape backup always has the letter after the last partition letter.

In addition, the MEGA PC supports LPT3:, and the COMn: designations are different. See the MEGA PC Supervisor's Guide for further details.

The filespec is different for communications devices. The filename is replaced with a list of options which specify certain parameters. Refer to OPEN "COM..." statement in Section 7 for more information.

Refer to the DOS manual for more detailed file specification and device information.

USER-INSTALLED DEVICE DRIVERS

GWBASIC allows you to use device drivers other than the standard ones supplied with the system software. For example, if you want to use a printer that has a different protocol from that recognized by the standard GWBASIC printer driver LPT1:, you can tell GWBASIC to use a different driver by specifying it in the OPEN statement. (Note that you will also have to tell DOS that the driver is installed by modifying the DEVICE command in the DOS configuration file CONFIG.SYS - see the DOS manual.) The OPEN statement is specified as follows:

```
OPEN filespec [FOR mode] AS [#]file number
   [LEN=rec length]
```

A user-installed device driver can be user-written or supplied by a third party. The following points should be borne in mind when a device driver is written:

1. The name of the driver must not end in a colon, since GWBASIC uses this to recognize predefined devices such as KYBD:, SCRN: etc. The only exception is that you can use the name LPT1: or LPT2: for a driver that replaces the standard printer driver.

2. The record length is set to 1 unless you change it by the LEN parameter of the OPEN statement. GWBASIC will buffer **rec length** characters before sending them to the driver.

3. GWBASIC only sends a carriage return (hex 0D) at the end of a line. If the device needs a line feed as well (hex 0A), the driver must provide it.

4. Device control information is passed from GWBASIC to the driver by the IOCTL statement, and from the driver to GWBASIC by the IOCTL$ function. The driver must be able to:

   a. set a maximum record length as specified in the OPEN statement

   b. return the current maximum record length to GWBASIC

   c. (for an input device) return an end-of-file condition to GWBASIC, so that a sequential input file that is open to a device driver can be closed if an INPUT statement tries to read beyond the last record in the file. If this happens, the device driver should return a Ctrl-Z, which is used by GWBASIC to generate the message "Input past end"

For more information about writing your own device drivers, see the DOS 2.0 or later version of the Microsoft MS-DOS Programmer's Reference Manual, Document No. 8411-200-00.

REDIRECTION OF INPUT AND OUTPUT

Normally a GWBASIC program takes its input from the keyboard, and outputs information to the screen so that you can see how the program execution is going. You may want to change this so that, for example, input is taken from data previously stored in a disk file, or output is sent straight to the printer instead of appearing on the screen. To cause this to happen, you have to redirect the input or output.

Redirection is specified when the GWBASIC command is used to invoke BASIC:

```
GWBASIC [<stdin] [[>]>stdout] ...
```

The two parameters **stdin** and **stdout** cause redirection of input and output respectively by specifying the name of a file or device from which input is read or to which output is written (note that if **stdout** is preceded by >> instead of >, output will be appended to the specified output file instead of overwriting it). The name can be any valid filename or filespec, or it can be a device identifier such as "LPT1:".

When input is redirected, all INPUT, LINE INPUT, INPUT$, and INKEY$ statements will read input from the file specified by **stdin** instead of from the keyboard. This will continue until an end-of-file marker (Ctrl-Z) is read from the input file (you can test for this condition using the EOF function). If the file has no such marker, or if a BASIC statement tries to read past end-of-file, any open files are closed, the message "Read past end" is displayed (or written to the output file if output is redirected) and BASIC terminates, passing control back to DOS.

If the ON KEY(**n**) statement is used when input is redirected, BASIC will continue to trap keys from the keyboard.

If you specify **stdin** as "KYBD:", input will continue to be read from the keyboard.

When output is redirected, information that would normally be displayed on the screen, such as the output of all PRINT statements, is sent to the file identified or device by **stdout**.

Using Ctrl-PrtSc will have no effect if output is redirected. Typing Ctrl-C or Ctrl-Break causes BASIC to terminate, passing control back to DOS.

If you specify **stdout** as "SCRN:", output will continue to be sent to the screen.

Error messages are sent to both the screen and the output file or device if only output is redirected; if both input and output are redirected, error messages are just sent to the output file or device.

The following examples illustrate some of the uses of redirection. The input statements referred to are INPUT, LINE INPUT, INPUT$ and INKEY$.

```
GWBASIC MYPROG>DATA.OUT
```

Data read by the input statements will continue to come from the keyboard. Data output by PRINT will go into the file DATA.OUT.

```
GWBASIC MYPROG<DATA.IN
```

Data read by the input statements will come from DATA.IN. Data output by PRINT will continue to go to the screen.

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

Data read by the input statements will now come from the file MYINPUT.DAT. Data output by PRINT will go into MYOUTPUT.DAT.

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

Data read by the input statements will now come from the file \SALES\JOHN\TRANS. Data output by PRINT will be appended to the file \SALES\SALES.DAT.

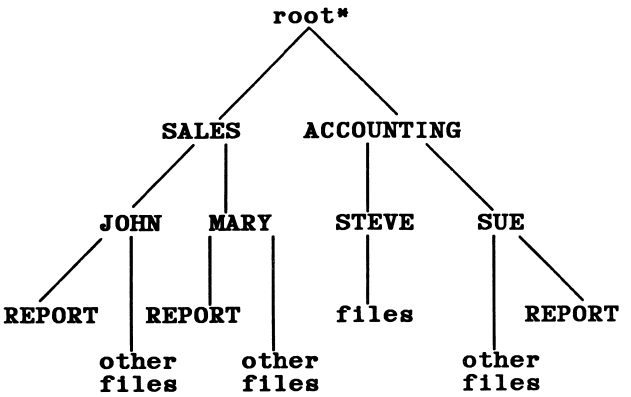
TREE-STRUCTURED DIRECTORIES

GWBASIC includes commands to enable you to organize your disk files in a tree-structured fashion, just as you can using DOS 2.0 and later releases, but without having to leave GWBASIC. The commands are the same as the DOS commands, namely:

MKDIR	creates a directory
CHDIR	specifies a different directory as the current directory
RMDIR	deletes a directory

Full details about tree-structured directories are given in the DOS manual.

Several of the examples given throughout Section 7 relate to a tree-structured disk file organization similar to this one:



*The root directory is designated by the initial "\" in a filespec.

To illustrate this use of tree-structuring, let us assume that the sales and accounting departments of a business share a computer that has a hard disk, and the individual employees use the system for preparing reports and maintaining accounting information. The files could be organized on the disk as in the example above. The disk contains a root directory (always identified by the name "\") which itself contains two further directories, called SALES and ACCOUNTING. Since these two are both accessed from the root, they are identified to DOS and GWBASIC by the names "\SALES"and "\ACCOUNTING".

In the same way, the SALES and ACCOUNTING directories contain further directories. For example, SALES contains two directories, called JOHN and MARY. These are the directories of individual employees, and are identified by the names "\SALES\JOHN" and "\SALES\MARY".

Going down to the lowest level of the structure, that of individual files, you can see for example that JOHN, in his directory, has various files, one of which is called REPORT. Assuming that the disk containing all these files is on drive C:, then this REPORT file can be identified to DOS and GWBASIC as:

```
"C:\SALES\JOHN\REPORT"
```

If the CHDIR command in DOS or GWBASIC has been used to access a particular directory (thus making it the **current directory**), an alternative way of identifying files and directories is possible. Let us assume that you have specified "CHDIR C:\SALES\JOHN" to make JOHN the current directory. Now you can identify the REPORT file in this directory in one of three ways:

```
REPORT
\SALES\JOHN\REPORT
..\JOHN\REPORT
```

Notice the last example, where the characters ".." replace "\SALES". The two periods are a shorthand method of specifying the current directory's **parent directory** (the directory at the next level up from the current one).

In the same way, you could refer to the file REPORT in the directory MARY in either of these ways (still assuming that JOHN is the current directory):

```
\SALES\MARY\REPORT
..\MARY\REPORT
```

To refer instead to the file REPORT in the directory SUE, which is not under SALES but under ACCOUNTING, you could specify one of the following from JOHN:

```
..\..\ACCOUNTING\SUE\REPORT
\ACCOUNTING\SUE\REPORT
```

Notice the multiple use of ".." in the first example. ".." takes you one level up the tree structure each time, so that in the example above, "..\.." takes you from JOHN first of all to SALES and then to the root, from where you can access ACCOUNTING.

Section 4
GRAPHICS

When you use GWBASIC for graphics with this microcomputer you are provided with an exclusive "super-resolution" capability. Super resolution allows the definition of 640 (horizontal) by 325 (vertical) positions on the 325-line system standard monochrome display, or 640 (horizontal) by 400 (vertical) positions on the 400-line system standard monochrome display.

In addition, medium- (320 x 200) and high- (640 x 200) resolution graphics are supported on the monochrome display. These lower resolution levels are standards for color systems and operate normally when using a color monitor with this computer. On the 325-line system standard monochrome display, medium resolution will appear in the upper left portion of the screen and high resolution will appear in the upper two-thirds of the screen. On the 400-line standard monochrome display, medium- and high-resolution graphics cover the entire screen (320 x 200 and 640 x 200 respectively). However, since high-resolution graphics supports only two colors, black and white, the only benefit the color/graphics monitor adapter adds is color capability to medium-resolution graphics and to text mode. (See the "SCREEN" statement in Section 7 for more information on screen modes.)

Points are always numbered from left to right and from top to bottom. If you think of the screen as a matrix of dots each having a vertical and horizontal location, you will understand the numbering system used. The horizontal position of each dot is x, and the vertical position is y. The upper left corner is therefore referred to as point 0,0, and the lower right corner point is either 639,324 (for 325-line systems) or 639,399 (for 400-line systems). The numbers 0,0 and 639,324 (or 639,399) are the coordinates of those particular points.

One of the features of GWBASIC is that you may have a number of graphics pages. The advantage of having multiple pages is that you can write to one page in memory while viewing a different page on the screen. The page in memory is called the **active page**, and the page on the screen is called the **visual page**. The maximum number of graphics pages depends on the type of system you have, as explained below. The default graphics page is the highest page number possible on your system.

For 325-line systems the maximum number of graphics pages is 8 (numbered 0 to 7). A 400-line system can have up to 16 pages (numbered 0 to 15), depending on the amount of memory available to DOS. Use the following formula to find the maximum page number for a 400-line system:

```
INT(M/32) - 1
```

where M is the amount of memory available to DOS in Kbytes (this value is displayed when the system is booted).

Pages 0 to 4 are reserved for system use, so you should use page 5 as the lowest number when specifying graphics pages.

CAUTION

When writing graphics programs, you should make sure that there will be no conflicts in memory usage between the program and the graphics. A CLEAR statement should be used at the beginning of the program to control the amount of workspace available to the program.

You will probably want to turn off the function key display using KEY OFF.

With the SCREEN statement, besides setting the resolution and the visual and active pages, you have options to set if you have an adapter for a color monitor (see Color/Graphics Monitor Adapter section below for more information). For advanced information about graphics, refer to Appendix B, which provides details on how the computer is configured for graphics, assembly language programming for graphics, and the graphics memory map.

The statements and functions that are used specifically for graphics are: CIRCLE, COLOR, DRAW, GET, LINE, PAINT, PMAP, POINT, PRESET, PSET, PUT, SCREEN, VIEW, VIEW PRINT and WINDOW.

Note that the COLOR statement if used with a monochrome monitor enables you to vary the image so that it blinks, reverses out, becomes invisible, highlighted, and/or underscored - see the COLOR Statement (Text) in Section 7.

For information on printing a graphics display screen, see the GRAPHICS command in the DOS manual.

HOW TO SPECIFY COORDINATES

There are two ways to use coordinates. When you specify a point as (x,y) you are using **absolute form**. Alternatively you can use the **relative form**, which is specified as STEP (x-offset, y-offset). For example, assume that the most recent point referenced was (x,y). The statement LINE STEP (10,5) would specify a point at offset 10 from x and offset 5 from y.

If the STEP option is used for the second coordinate in the statement, it is relative to the first coordinate. For example, the statement LINE (10,15)-STEP(20,30) would draw a line from point (10,15) to point (30,45).

COLOR/GRAPHICS MONITOR ADAPTER

This adapter is a card that fits in one of the expansion slots on the main PCB. The adapter is used if you have a color monitor, allowing you to add color to either text or graphic images.

In text mode you have the ability to use 16 colors (see COLOR Statement (Text)). In addition, you have the opportunity for multiple pages of text.

Medium resolution (320 by 200 points) gives you a combination of colors: a palette of 3 colors associated with the color number you have chosen. See COLOR Statement (Graphics) for details.

High resolution (640 by 200 points) has only two colors, black and white.

Super resolution is not supported by the color/graphics adapter.

Section 5
PROGRAMMING CONCEPTS

This section introduces the raw materials used to create a BASIC program, namely:

- character set
- constants
- variables
- numeric precision
- arrays
- expressions
- operators

CHARACTER SET

The GWBASIC character set consists of alphabetic characters, numeric characters, and special characters.

The **alphabetic characters** are the upper-case and lower-case letters of the alphabet. The **numeric characters** are the digits 0 through 9.

All of the remaining characters on the keyboard are **special characters**, some of which also have special meanings to GWBASIC. For example, the "-" character can be just a character as you might use it on a typewriter. On the other hand, in a BASIC program statement, the same character can be an instruction to perform a subtraction.

Table 5-1 lists the special characters, together with any special meaning that they have to GWBASIC. Table 5-2 lists certain special characters that can be generated in more than one way.

Another set of characters includes those generated in conjunction with the Ctrl (control) key. The control characters generally do not cause individual characters to be displayed, although they may have very noticeable effects on the screen. For example, Ctrl and L will cause the screen to be cleared. These characters are listed in Table 2-1.

The Alt key can be used to generate certain GWBASIC keywords, for example typing Alt-A causes the word AUTO to be displayed. This saves the trouble of typing these keywords in full each time you want to use them. The full list of keywords is given in Table 5-3.

A final set of special characters includes those which can be generated directly on the screen by pressing the Alt key and entering a number in the range from 128 through 255. When the Alt key is released, the character will appear. A complete list of the character set can be found in Appendix C, ASCII Character Codes.

Table 5-1

SPECIAL CHARACTERS

Character	Explanation
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Multiplication symbol
/	Division symbol
\	Integer division symbol
^	Exponentiation symbol
%	Declaration character for integer variables
#	Declaration character for double-precision variables
$	Declaration character for string
!	Declaration character for single-precision variables
'	(Apostrophe) remark delimiter
:	Program statement separator
?	Abbreviation for PRINT statement
"	String delimiter
(	Left parenthesis
)	Right parenthesis
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
;	Semicolon
<	Less-than sign
>	Greater-than sign
@	At sign
_	Underscore

Certain characters have a particular effect, and most of these can be generated in more than one way, namely by:

- pressing the key itself
- pressing Ctrl and another key
- holding down Alt and typing a numerical sequence
- using CHR$ with the appropriate value in a GWBASIC program

Table 5-2 gives details of these alternatives.

Table 5-2

OTHER SPECIAL CHARACTERS

Key	Ctrl	Alt	CHR$	Description
	C or Break			Stop program execution and return to BASIC command level.
	G	007	7	Generate a beep sound.
Back Space	H	008	8	Delete character to the left of cursor and move cursor to that position.
Tab	I	009	9	Move cursor 8 spaces to the right.
	S or Num Lock			Suspend program execution. Resume program execution after after Ctrl-S or Ctrl-Num Lock.

Table 5-2 (Cont.)

Key	Ctrl	Alt	CHR$	Description
Esc	U or [		27	Erase entire logical line.
	→			Move cursor to start of next word.
	N			Move cursor to end of logical line.
	← or B			Move cursor to start of previous word.
	Home	011	11	Move cursor to upper left corner of screen.
	L	012	12	Clear screen.
	E or End			Erase to end of logical line.
Insert	R			Toggle insert mode.
Return or Enter	M	013	13	Carriage return (end a logical line).

There is also a set of keywords generated by the Alt key. They offer a shorthand way to refer to certain statements, commands, or functions in GWBASIC. Table 5-3 lists these.

Table 5-3

ALT KEYWORDS

Alt	Keyword	Alt	Keyword	Alt	Keyboard
A	AUTO	J	(no word)	S	SCREEN
B	BSAVE	K	KEY	T	THEN
C	COLOR	L	LOCATE	U	USING
D	DELETE	M	MID$	V	VAL
E	ELSE	N	NEXT	W	WIDTH
F	FOR	O	OPEN	X	XOR
G	GOTO	P	PRINT	Y	(no word)
H	HEX$	Q	(no word)	Z	(no word)
I	INPUT	R	RUN		

CONSTANTS

Constants are the actual values GWBASIC uses during execution. There are two types of constants: string (character) and numeric.

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

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

Numeric constants are positive or negative numbers. Note that a plus sign is optional on a positive number. Numeric constants in GWBASIC may not contain commas. There are five types of numeric constants:

1. Integer

   Whole numbers between -32768 and 32767. These constants do not contain decimal points.

2. Fixed-point

   Positive or negative real numbers, i.e. numbers that contain decimal points.

3. Floating-point

   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). Double-precision floating point constants use the letter D instead of E. For more information refer to "Numeric Precision" below. (The E or D means "times ten to the power of.")

   Examples:

   235.988E-7 = .0000235988
   2359E6 = 2359000000

   The range for these is 10E-38 to 10E+38 (positive or negative).

4. Hexadecimal

   Hexadecimal numbers with up to four digits and the prefix &H. Hexadecimal digits are the numbers 0 through 9, A, B, C, D, E, and F.

   Examples:

   &H76
   &H32F

5. Octal

   Octal numbers with up to 6 digits and the prefix &O or &. Octal digits are 0 through 7.

   Examples:

   &O347
   &1234

Numeric Precision

Numbers may be stored as either integers, single precision, or double precision. Constants entered in integer, hexadecimal or octal format are stored in two bytes of memory and are interpreted as integers or whole numbers. In single precision up to 7 digits may be stored and printed, but only 6 will be accurate. In double precision, numbers may be stored with 16 digits of precision, and 16 digits may be printed.

A single-precision constant is any numeric constant that is not an integer and has one of the following characteristics:

1. Seven or fewer digits.
2. Exponential form using E.
3. A trailing exclamation point.

A double-precision constant is any numeric constant that has one of these characteristics:

1. Eight or more digits.
2. Exponential form using D.
3. A trailing number sign.

Examples:

46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

VARIABLES

Variables are names used to represent values in a GWBASIC program. There are two types: numeric and string. A numeric variable always has a value that is a number. All other variables consist of strings of characters. The variable type must match the type of data being assigned to it.

The value of a variable may be set as a constant, or it may be assigned as the result of calculations or various data input statements in the program.

If a variable is used before a value is assigned to it, its value is assumed to be zero until a value is assigned.

How to Name a Variable

GWBASIC variable names may be any length within a meaningful program line of up to 250 characters. Up to 40 characters of the name are significant. Variable names can contain letters, numbers, and the decimal point. However, the first character must be a letter. Special ending characters are used to identify types of variables. See "How to Declare Variable Types" below.

A variable name must not be a reserved word, although a reserved word can be embedded as part of a variable name (for example, TAN cannot be used but TANGENT is a valid name). The one exception is that no variable may begin with the characters USR. Reserved words consist of all GWBASIC commands, statements, function names, and operator names. A complete list is provided in Appendix D.

If a variable begins with FN, it is assumed to be a call to a user-defined function. (Refer to DEF FN statement in Section 7.)

How to Declare Variable Types

String variable names are written with a dollar sign as the last character. For example:

```
A$ = "SALES REPORT"
```

The dollar sign announces that the variable will represent a string. Storage requirements are 3 bytes plus the length of the string.

Numeric variable names may declare integer, single-, or double-precision values. The type declaration characters for these as well as the number of bytes required to store each type of value are as follows:

% Integer variable (2 bytes)

! Single-precision variable (4 bytes)

Double-precision variable (8 bytes)

If the variable type is not declared, the default is single precision.

Note that double-precision variables require twice the storage space of single-precision variables. They also require more time for arithmetic operations. Integer variables produce the fastest and most compact object code.

Examples of GWBASIC variable names:

PI#	double precision
MINIMUM!	single precision
LIMIT%	integer
N$	string
ABC	single precision

Another way to declare variable types is through the following statements: DEFINT, DEFSTR, DEFSNG, and DEFDBL. Refer to DEFtype statements in Section 7 for further information.

Array Variables

An array is a list or matrix table of numeric or string values.

An array is created by establishing dimensions for a variable (refer to DIM statement in Section 7). Each value in an array is called an **element** and is identified by means of a **subscript** attached to the variable name.

```
DIM V$(4,4,2)
```

The preceding statement creates a three-dimensional array of string values. The dimensions might be thought of as rows, columns, and pages. The sequence is for the user to define. The statement has established the maximum value for subscripts for the array. The subscripts must be positive integer expressions.

```
A$=V$(2,1,1)
```

The preceding assigns the value of an element of the array to A$.

If an array element is created without a DIM statement, a single-dimension array is implicitly created with a maximum subscript of 10.

The minimum value for a subscript is 0 unless it is set to 1. If you do not intend to use the 0 element in an array, you can save data storage space by using the OPTION BASE statement (refer to Section 7). Setting the minimum to 1 would save 8008 bytes in the following array.

```
AB#(1000,2)
```

The maximum number of dimensions for an array is 255. Up to 32,767 elements can be specified per dimension. The maximum amount of memory that can be occupied by an array is 64K.

HOW GWBASIC CONVERTS NUMBERS FROM ONE PRECISION TO ANOTHER

The following rules apply when GWBASIC converts a number from one precision to another.

1. If a numeric constant of one type is assigned to a numeric variable of a different type, the number will be stored as the type declared in the target variable name.

   Example:

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

   Note that if a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.

2. When an expression is evaluated, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e. that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

   Examples:

   ```
   10 D# = 6#/7
   20 PRINT D#
   RUN
    .8571428571428571
   ```

The arithmetic was performed in double precision, and the result was returned in D# as a double-precision value.

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

The arithmetic was performed in double precision, and the result was returned to D (single-precision variable), rounded, and printed as a single-precision value.

3. Logical operators (see below) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

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

   Example:

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

5. Precision is not increased when converting from a lower- to a higher-precision number. For example if a single-precision value (A) is assigned to a double-precision variable (B#), only the first six digits of B# will be valid because only six digits of accuracy were supplied with A.

   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.

   Example:

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

6. When converting from a higher-precision value to a lower-precision variable, the result is rounded.

Example:

```
10 C = 55.8834567#
20 PRINT C
RUN
 55.88346
```

This affects assignment statements as well as function and statement evaluations.

EXPRESSIONS AND OPERATORS

An expression may be a string or numeric constant or variable. An expression may also combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on numeric as well as string values. They may be divided into the following categories: arithmetic, relational, logical, and functional. Each is described below.

Arithmetic Operators

The arithmetic operators are listed in Table 5-4 in order of precedence, i.e., when several arithmetic operations take place in the same statement, the operation highest in the table will be performed first. If two or more operations have the same level of precedence, the leftmost operation will be performed first. Note that you can change the order of evaluation by using parentheses. Operations within parentheses are performed first. Within parentheses the normal order of operations is maintained.

Table 5-4

ARITHMETIC OPERATORS

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, floating-point division	X*Y X/Y
\	Integer division	X\Y
MOD	Modulo arithmetic	X MOD Y
+,-	Addition, subtraction	X+Y X-Y

Integer division is denoted by the backslash (\). The operands are rounded to integers (in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.

Example:

```
10 A = 10\4
20 B = 25.68\6.99
30 PRINT A;B
RUN
 2 3
```

Modulo arithmetic is denoted by the operator MOD. It yields the integer remainder of an integer division.

Example:

```
10 A = 10 MOD 4
20 PRINT A
RUN
 2
```

Remainder 2 results when 10 is divided by 4.

```
PRINT 25.68 MOD 6.99
5
```

Remainder 5 results when 26 is divided by 7.

To change the order in which operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Some sample algebraic expressions and their GWBASIC counterparts are given in Table 5-5.

Table 5-5

SAMPLE ALGEBRAIC EXPRESSIONS AND THEIR GWBASIC COUNTERPARTS

Algebraic Expression	GWBASIC Expression
x+2y	X+Y*2
$x-\frac{y}{z}$	X-Y/Z
$\frac{xy}{z}$	X*Y/Z
$\frac{x+y}{z}$	(X+Y)/Z
$(x^2)^y$	(X^2)^Y
x^{y^z}	X^(Y^Z)
x(-y)	X*(-Y)

Note in the last example that two consecutive operators must be separated by parentheses.

If, during the evaluation of an expression, division by zero occurs, a "Division by zero" error message is displayed. Machine infinity (a number recognizable by the fact that it ends with "E+38") with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation operator 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.

If there is an overflow during computation, an "Overflow" error occurs, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

Relational Operators

Relational operators are used to compare two values, which may be either both numeric or both string. The result of the comparison is either "true" (-1) or "false" (0). This result is usually used to make a decision regarding program flow. (See IF statement in Section 7.) Relational operators are listed in Table 5-6.

Table 5-6

RELATIONAL OPERATORS

Operator	Relation Tested	Example
=	Equality	X=Y
<> or ><	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<= or =<	Less than or equal to	X<=Y X=<Y
>= or =>	Greater than or equal to	X>=Y X=>Y

The equal sign is used to assign a value to a variable. Refer to LET statement in Section 7.

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

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

is true if the value of X plus Y is less than the value of T-1 divided by Z.

Example:

```
IF SIN(X)<0 GOTO 1000
```

GWBASIC must evaluate SIN(X) first to determine if it is less than zero.

Strings are compared by taking one character at a time from each and comparing their ASCII codes. (Refer to Appendix C, ASCII Character Codes, for a complete list of these codes.)

As long as the ASCII codes are the same, the strings are equal. When the codes differ, the string with the lower number precedes (i.e., is regarded as less than) the one with the higher number. If during the comparison the end of one string is reached, the shorter string precedes the longer one.

Leading and trailing blanks are significant.

The following examples of relational expressions are all true, i.e., the result of the relational operation is -1.

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/84"                    (where B$ = "8/12/84")
```

Note that all string constants used in comparison expressions must be enclosed in quotation marks.

Logical Operators

Logical operators perform tests on numeric values. These Boolean operations are usually used to make decisions by connecting two or more relations and returning a true or false value. (Refer to IF statement in Section 7 for more information.)

The result of a logical operation is a number that is "true" if it is not zero, or "false" if it is equal to zero. Table 5-7 lists the results of these operations ("1" indicates a true value, and "0" indicates a false value). The values are shown in order of precedence. Thus if several operations take place in the same statement, NOT will be performed before AND, etc. If two or more operations have the same level of precedence, the leftmost one will be performed first.

In an expression, logical operations are performed after arithmetic and relational operations.

Examples:

```
IF D<200 AND F<4 THEN 80
```

The result will be true if the value of D is less than 200 and the value of F is less than 4; if both these conditions are satisfied the program will jump to line 80.

```
IF I>10 OR K<0 THEN 50
```

The result will be true if I is greater than 10, or K is less than 0, or both. Thus if either or both of these conditions are satisfied the program will jump to line 50.

Logical operators work by converting 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 "Overflow" error results.) The given operation is performed on these integers bit by bit, and the result is determined by the corresponding bits in the two operands.

Table 5-7

RESULTS OF LOGICAL OPERATIONS IN BASIC

Operation	Value	Value	Result	Terminology
NOT	X		NOT X	Logical
	1		0	Complement
	0		1	
AND	X	Y	X AND Y	Conjunction
	1	1	1	
	1	0	0	
	0	1	0	
	0	0	0	
OR	X	Y	X OR Y	Disjunction
	1	1	1	
	1	0	1	
	0	1	1	
	0	0	0	
XOR	X	Y	X XOR Y	Exclusive
	1	1	0	OR
	1	0	1	
	0	1	1	
	0	0	0	
EQV	X	Y	X EQV Y	Equivalence
	1	1	1	
	1	0	0	
	0	1	0	
	0	0	1	
IMP	X	Y	X IMP Y	Implication
	1	1	1	
	1	0	0	
	0	1	1	
	0	0	1	

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port, and the OR operator may be used to "merge" two bytes to create a particular binary value.

The following examples will help demonstrate how the logical operators work.

63 AND 16 results in 16

Since 63 is binary 111111 and 16 is binary 10000, 63 AND 16 equals 010000 in binary, which is equal to 16.

-1 AND 8 results in 8

Since -1 is binary 11111111 11111111 and 8 is binary 1000, -1 AND 8 equals binary 00000000 00001000, or 8.

4 OR 2 results in 6

Since 4 is binary 100 and 2 is binary 10, 4 OR 2 is binary 110, which is equal to 6.

NOT X = -(X+1)

The two's complement of any integer is the bit complement plus one.

If both operands are equal to either 0 or -1, a logical operator will return either 0 or -1.

Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. Certain functions, such as SQR (square root) or SIN (sine), reside in GWBASIC. Details of these functions are given in Section 7.

GWBASIC also allows you to define your own functions with the DEF FN statement. (Refer to this statement in Section 7.)

String Operators

String operators include concatenation and string functions. Strings may be concatenated, or joined together, by using +. For example:

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

String functions return results which are strings. All the functions listed in Section 7 which end in "$" are string functions. In addition, the DEF FN statement can be used to define string as well as numeric functions.

Section 6
COMMUNICATIONS

This section describes the GWBASIC statements required to support RS232 asynchronous communication with other computers and peripherals.

OPENING A COMMUNICATIONS BUFFER

OPEN "COM..." allocates a buffer for communications I/O. Opening a communications buffer in this way is the equivalent of using OPEN to open a data file on disk.

COMMUNICATIONS I/O

Since the communications buffer is opened as a file, all input/output statements valid for disk files are valid for COM.

Sequential input statements for communications are the same as those for disk files. They are:

INPUT #file number
LINE INPUT #file number
INPUT$

Communications sequential output statements are also the same as those for disk files, and are:

PRINT #
PRINT # USING
WRITE #

Refer to these statements in Section 7 for details of coding format and usage.

Communications I/O Functions

The most difficult aspect of asynchronous communication is being able to process characters as fast as they are received. At rates above 2400 bps, it is necessary to suspend character transmission from the host long enough to "catch up". This can be done by sending XOFF (Ctrl-S) to the host and XON (Ctrl-Q) when ready to resume.

Three functions are provided to help in determining when an "over-run" condition is imminent. These are:

LOC(**x**)

Returns the number of characters in the input buffer waiting to be read. If that number is greater than 255, LOC returns 255.

LOF(**x**)

Returns the number of free bytes in the input buffer. This is the same as **n**-LOC(**x**), where **n** is the size of the communications buffer as set by the /C: option of the GWBASIC command. The default for **n** is 256.

EOF(**x**)

Returns true (-1) if the input buffer is empty and false (0) if any characters are waiting to be read.

A "Communications buffer overflow" error can occur if a read is attempted after the input buffer is full (i.e., LOC(**x**) returns 0).

INPUT$ Function for COM Files

The INPUT$ function is preferred over the INPUT and LINE INPUT statements when reading COM files, since all ASCII characters may be significant in communications. INPUT is least desirable because input stops when a comma or carriage return is seen. LINE INPUT terminates when a carriage return is seen.

INPUT$ allows all characters read to be assigned to a string. INPUT$ (**n**,**y**) will return **n** characters from file **y**. The following statements are efficient for reading a COM file:

```
10 WHILE NOT EOF(1)
20 A$=INPUT$(LOC(1),#1)
    .
    .
    .
(program processes data returned in A$)
    .
    .
    .
60 WEND
```

The statements above return the number of characters in the buffer and store them in A$. If there are more than 255 characters, only 255 will be returned at a time to prevent "String overflow". Furthermore, if this is the case, EOF(1) is false, and input continues until the buffer is empty. This code enables fast processing.

GET and PUT for COM Files

GET and PUT for COM files are slightly different from disk files and are used for fixed-length I/O from or to the COM file. Instead of specifying the record number, you specify the number of bytes to be transferred into or out of the file buffer. This number cannot exceed the value set by the LEN option on the OPEN "COM..." statement. Refer to GET and PUT in Section 7.

A SAMPLE PROGRAM

The following program enables the computer to be used as a conventional terminal. Besides full duplex communication with a host, the program allows data to be "down-loaded" to a file. Conversely, a file may be "up-loaded" (transmitted) to another terminal.

In addition to demonstrating the elements of asynchronous communication, this program should be useful in transferring GWBASIC programs and data to and from the PC.

Note that this program is set up to communicate with a DEC-20 using XON and XOFF. You may want to modify it for your environment.

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$(80," ")
 40 FALSE=0:TRUE= NOT  FALSE
 50 MENU=5 ' Value of MENU Key (Ctrl-E)
 60 XOFF$=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,7,1,cs,ds,cd,1f"
130 OPEN COMFIL$ AS #1
140 OPEN "SCRN:" FOR OUTPUT AS #2

200 LOCATE 25,1: PRINT "ASYNC TTY PROGRAM";SPC(5);
    :COLOR 0,7:PRINT"TERMINAL EMULATION MODE";
    :COLOR 7,0:PRINT SPC(5);"TYPE CTRL-E FOR MENU"
    :LOCATE 2,1
205 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 #2,A$;:IF LOC(1)>0 THEN 240
270 IF PAUSE THEN PAUSE=FALSE:PRINT #1,XON$;
280 GOTO 210

300 LOCATE 25,1: PRINT "ASYNC TTY PROGRAM";SPC(5);
    :COLOR 0,7:PRINT"FILE TRANSFER MODE";:COLOR
    7,0:PRINT STRING$ (30," "):LOCATE 2,1
305 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
    #3:GOTO 1000
430 OPEN DSKFIL$ FOR OUTPUT AS #3
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$(LOC(1),#1)
530 PRINT #3,A$;: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 #3:CLS:LOCATE 1,1:PRINT "* Download
    complete *"
650 GOTO 200

1000 WHILE NOT EOF(3)
1010 A$=INPUT$(1,#3)
1020 PRINT #1,A$;
1030 WEND
1040 PRINT #1,CHR$(26); 'CTRL-Z to make close file.
1050 CLOSE #3:CLS:LOCATE 1,1:PRINT "** Upload
     complete **"
1060 GOTO 200

9999 CLOSE:KEY OFF
```

Notes on the TTY program:

Line No.	Comments
10	Sets the screen to monochrome text mode and sets the width to 80.
15	Turns off the function key display, clears the screen, and makes sure that all files are closed. Asynchronous implies character I/O as opposed to line or block I/O. Therefore, all PRINT statements (either to the communications file or to the screen) are terminated with a semicolon. This suppresses the carriage return and line feed normally issued at the end of a PRINT statement.
20	Defines all numeric variables as integers. This primarily benefits the subroutine at lines 600-620. To optimize speed always use integer counters in loops where possible.

Line No.	Comments
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.
100-200	Prints program name, asks for Baud rate (speed). Opens communication to file number 1, even parity, 7 data bits. Programmer exercise: Modify this section to check for valid baud rates before continuing.
205-280	This section performs full duplex I/O between the screen and the device connected to the RS232 connector as follows: 1. Reads a character from the keyboard into A$. Note that INKEY$ returns a null string if no character is waiting. 2. If no character is waiting, checks whether any characters are being received. 3. If the character at the keyboard is the menu key, a file can be downloaded. Gets file name. 4. If character (A$) is not the menu key, then sends it by writing to the communications file (PRINT #1,A$). 5. At 230 checks whether characters are waiting in buffer. If not, goes back and checks keyboard. 6. At 240, if more than 128 characters are waiting, sets pause flag to suspend input. Sends XOFF to stop further transmission. 7. At 250-260, reads and displays contents of buffer on screen until empty. Continues to monitor size of buffer (in 240). Suspends transmission if falling behind.

Line No.	Comments
	8. Resumes transmission by sending XON only if suspended by previous XOFF. Repeats process until menu key is struck.
300-310	Gets disk file name to download to. Opens file to tie number 2.
400-430	Asks if file named is to be transmitted (uploaded) or received (downloaded).
440	Sends a carriage return to the host to begin downloading. 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: **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), performs a time-out routine (explained in line 600).
510	Again, if more than 128 characters are waiting, signals a pause and sends XOFF to the host.
520-530	Reads all characters in the buffer (LOC(x)) and writes them to disk (PRINT #3..) until caught up.
540-550	If a pause was issued, restarts host by sending XON and clears the pause flag. Continues process until no characters are received for a pre-determined time.
600-650	Time-out subroutine. The FOR loop count is determined by experimentation. That is, if no character is received from the host for 17-20 seconds, transmission is assumed to be complete. If any character is received during this time (line 610), sets I well above FOR loop range to exit loop and return to caller. If host transmission is complete, closes the diskette file and returns to being a terminal.

Line No.	Comments
1000-1060	Transmit routine. Reads one character into A$ with INPUT$ statement. Sends character to device in line 1020. Sends Ctrl-Z in line 1040 in case receiving device needs one to close its file. Finally, in lines 1050 and 1060, closes disk file, prints completion message, and returns to conversation mode in line 200.
9999	Not executed in version of program as shown above. This line closes the communications file left open and restores the function key display. Programmer exercise: Add some lines to the routine in lines 400-420 to exit the program via line 9999.

OPERATION OF CONTROL SIGNALS

The following sections give more detailed technical information about communicating with another computer or peripheral from BASIC. This information may only be of interest to advanced programmers.

Output from the asynchronous communications (serial) port conforms to the EIA RS-232C standard. Therefore control signals transmitted or received are DC voltages that are either ON (greater than +3 volts) or OFF (less than -3 volts).

Control of Output Signals with OPEN

When GWBASIC is invoked the RTS (Request To Send) and DTR (Data Terminal Ready) transmission lines are both turned off. With an OPEN "COM..." statement both these lines are turned on. However you may suppress the RTS signal by using the RS option on the OPEN "COM..." statement. Lines stay on until the communications file is closed (with CLOSE, END, NEW, RESET, SYSTEM, or RUN without the R option).

The DTR line is on and stays on even if the OPEN "COM..." statement fails with an error. This allows you to retry the OPEN without having to execute a CLOSE.

Use of Input Control Signals

If either the CTS (Clear To Send) or DSR (Data Set Ready) lines are OFF, an OPEN "COM..." statement will not execute and after one second, BASIC will return a "Device Timeout" error.

The Carrier Detect (also called Receive Line Signal Detect or RLSD) line can be either on or off. It will have no effect on the operation of the program.

Use the RS, CS, DS, and CD options on the OPEN "COM..." statement to specify how these lines should be tested. This information is given in the OPEN "COM..." statement in Section 7.

If signals being tested are turned off while a program is executing, I/O statements associated with the communications file will not work. For example, if you turn the CTS or DSR line off and subsequently execute a PRINT # statement, you will get a "Device Fault" or "Device Timeout" error. However, the RTS and DTR stay on even if such an error occurs.

Test for a line disconnect by using INP to read the bits in the Modem Status Register on the asynchronous communications adapter (for information on using INP and OUT to read from and write to the 8250 UART registers, see "Accessing the Registers" below). With the built-in communications adapter use INP(&H3FE) to read the register, and with an add-on adapter use INP(&H2FE). Use the delta bits in the status register to determine if transient signals have appeared on any of the control lines. Remember that for a control signal to have meaning, the pin corresponding to that signal must be connected in the cable to your modem or to another computer.

Another way to test for bits is with the Line Status Register. Use INP(&H3FD) with the built-in communications adapter and INP(&H2FD) with an add-on communications adapter to access this register. The bits can be used to determine what types of errors have occurred on receipt of characters from the communications line or whether a break signal has been detected.

Direct Control of Output Control Signals

Use the OUT statement to control RTS or DTR control signals directly. Whether these signals are on or off is controlled by bits in the Modem Control Register, which is at address 3FC (in BASIC, &H3FC, to indicate a hexadecimal number).

CAUTION

Take great care when modifying hardware registers directly as incorrect programming can cause system degradation or failure.

The Line Control Register may also be used to modify bits. Use care with this method since most of the bits in this register have been set by BASIC at the time an OPEN statement is executed and changing any of them could cause communications failure. This register is at address 3FB (in BASIC, &H3FB).

First read the register with INP, and then rewrite it by changing only the pertinent bit or bits.

Bit 3, the Parity Enable bit, in the Line Control Register indicates whether parity checking is enabled or not. To check the status of this bit while a program is running, use the following code in the program:

```
10 DEFINT S
20 S=INP (&H3FB) 'read the line control register
30 IF (S AND 8) THEN GOTO 60
40 PRINT "Parity currently disabled"
50 GOTO 70
60 PRINT "Parity currently enabled"
70      .
        .
        .
```

For further information, consult the Technical Reference Manual for your system.

COMMUNICATIONS ERRORS

Communications errors can occur at several different stages: on opening the communications file, when reading data or when writing data.

An error message that may occur when opening the communications file with OPEN "COM..." is:

"Device Timeout"

Occurs if a signal to be tested (CTS, DSR, or CD) is missing.

Messages that can be displayed in the event of an error when reading data are:

"Com buffer overflow"

Output if overrun occurs.

"Device I/O error"

Indicates overrun, break, parity, or framing errors.

"Device Fault"

Indicates loss of DSR or CD signal.

Possible errors when writing data are:

"Device Fault"

Caused by loss of CTS, DSR, or CD on a Modem Status Interrupt while BASIC was performing other processing.

"Device Timeout"

Indicates loss of CTS, DSR, or CD while waiting to write data to the output buffer.

ACCESSING THE REGISTERS

You may read from or write to any of the 8250 UART registers via the CPU. These registers are used to control 8250 operations and to transmit and receive data. The registers are the Modem Status Register (MSR), Modem

Control Register (MCR), Interrupt Enable Register (IER), Interrupt Identification Register (IIR), Line Status Register (LSR), Receiver Buffer Register (RBR), Transmitter Holding Register (THR) and Line Control Register (LCR). Full details are given in the Technical Reference Manual for your system.

To read the contents of one of these registers, use the INP function. For example, INP(&H3FB) returns the contents of the Line Control Register.

To write to one of these registers, use the OUT statement. For example, OUT &H3FB,&H40 writes the hexadecimal value 40 (binary 0100 0000) to the Line Control Register, causing bit 6 (Set Break) to be set.

Section 7

GWBASIC COMMANDS, STATEMENTS, FUNCTIONS, AND VARIABLES

INTRODUCTION

Following is an alphabetical listing of the commands, statements, functions, and variables in GWBASIC. Commands are ways to tell GWBASIC to perform an action immediately. Statements are used to define parameters you wish to set. Executable statements tell GWBASIC what to do next, and nonexecutable statements cause no action. An example of a nonexecutable statement is REM, which allows you to insert a remark in the program you are writing. Functions return a numeric or string result. Variables represent values used in a GWBASIC program.

In the specifications that make up this section, the format line shows you how to enter the command, statement, function, or variable. Remember the following rules:

- Words in capital letters must be entered as shown. Note that GWBASIC will convert words to upper case unless they are part of a quoted string, remark, or DATA statement.
- You supply the information in lowercase bold letters.
- Square brackets [ ] enclose information that is optional.
- Ellipses (...) mean that the preceding item may be repeated as often as necessary.
- Braces { } indicate a choice between two or more items, which are separated by vertical bars |
- All punctuation except that in square brackets must be included and must be in the position indicated by the format.

The following is an example of a format line:

```
INPUT[;]["prompt";] variable[,variable]...
```

In this statement the keyword INPUT may be followed by a semicolon. Then a prompt may be included in quotation marks followed by a semicolon. At least one variable is required for the statement, but others may be added if they are separated by commas.

Note the following definitions of parameters that appear throughout this section.

filespec = a string expression conforming to the rules for file specification given in Section 3 of this manual

numvar = the name of a numeric variable

variable = the name of any variable, numeric or string

Note also that while the format for functions is:

```
v = function
```

You may also use the following direct command:

```
PRINT function
```

The purpose of the command, statement, function, or variable is provided next, after which there is amplifying material telling more about each. Examples are also given and explained.

A complete summary of the GWBASIC commands, statements, functions and variables is given in Appendix I.

NOTE: For the MEGA PC, any code that references the speaker or the music queue will have no effect. This includes the PLAY function and the BEEP, ON PLAY(n), PLAY [ON/OFF/STOP] and SOUND statements.

ABS Function

FORMAT `v = ABS(x)`

PURPOSE Returns the absolute value of `x`.

■ ■ ■

`x` may be any numeric expression.

```
PRINT ABS(8*(-3))
 24
Ok
```

Since the absolute value of a number is always positive or zero, in this example it is positive 24.

ASC Function

FORMAT **v** = ASC(**a$**)

PURPOSE Returns the ASCII code for the first character of a string.

■ ■ ■

a$ may be any string expression.

See Appendix C for ASCII codes.

If **a$** is null, an "Illegal function call" error is returned.

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

The above shows the ASCII code for "T" is 84. PRINT ASC("TEST") would show the same result.

ATN Function

FORMAT v = ATN(x)

PURPOSE Returns the arctangent of x in radians.

■ ■ ■

The expression x may be any numeric type, but the default evaluation of ATN is performed in single precision. The evaluation will be in double precision if the /D switch is specified in the GWBASIC command.

The result is a value in the range -pi/2 to pi/2, where pi=3.141593.

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

The above shows how the arctangent of 3 is calculated.

AUTO Command

FORMAT AUTO **[line number[,increment]]**

PURPOSE Generates a line number automatically every time <Return> is pressed.

■ ■ ■

line number is the number of the beginning line.

increment is the value that will be added to each line number to get the next one.

AUTO is especially useful when you are entering programs because it spares you the trouble of having to type each line number.

AUTO begins at **line number** and increments each subsequent line number by **increment**. The default for both values is 10. If **line number** is followed by a comma but **increment** is not specified, the last increment specified in an AUTO command is assumed. If **line number** is omitted but **increment** is included, then the first line number is 0.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn you that any input will replace the existing line. However, pressing <Return> immediately after the asterisk will save the existing line, and AUTO will generate the next line number.

AUTO ends when you press Ctrl-Break or Ctrl-C. The line in which Ctrl-Break or Ctrl-C is typed is not saved, and you are returned to command level.

NOTE: When in AUTO mode, you may change only the current line. If you want to change another line, you must exit AUTO.

```
AUTO 100,50
```

Generates line numbers 100, 150, 200 ...

```
AUTO
```

Generates line numbers 10, 20, 30, 40 ...

BEEP Statement

FORMAT BEEP

PURPOSE Sounds the speaker.

■ ■ ■

BEEP sounds the speaker at 800 Hz for 1/4 second.

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

```
2430 IF X < 20 THEN BEEP
```

If X is less than 20, the speaker will sound.

BLOAD Command

FORMAT BLOAD **filespec[,offset]**

PURPOSE Loads a file in binary format into memory.

■ ■ ■

filespec identifies the file to be loaded, and has the form shown under "File and Device Information" in Section 3.

offset is a numeric expression in the range 0 to 65535. Loading starts at this address and is specified as an offset into the segment declared by the last DEF SEG statement.

If **offset** is omitted, the offset and segment address are assumed to be those specified by a previous BSAVE command. In this case the file is loaded into the same location from which it was saved.

If the device is omitted, the DOS default drive is assumed.

WARNING

BLOAD does not perform an address range check. It is therefore possible to load a file anywhere in memory. Make sure that the file does not load over GWBASIC's stack, GWBASIC's variable area, or your GWBASIC program (see the memory map in Appendix G). Careful use of the /M switch in the GWBASIC command will help in avoiding this problem.

BLOAD and BSAVE are often used for loading and saving assembly language programs. (See the CALL statement for how to execute assembly language programs from within a GWBASIC program.) However, any segment may be specified as the source or target for these statements by means of the DEF SEG statement. This allows you to save and display screen images by saving from or loading to the screen buffer. The memory address for any given super-resolution graphics page is:

```
(page * 800H):0000
```

You will need to input this address in a DEF SEG statement prior to using BLOAD with screen images.

```
10 'Load subroutine at 6000:F000
20 DEF SEG=&H6000 'Set segment to 6000 Hex
30 BLOAD"PROG1",&HF000 'Load PROG1
```

The segment address is set at 6000 hex and loads PROG1 at F000.

BSAVE Command

FORMAT BSAVE **filespec,offset,length**

PURPOSE Transfers the contents of the specified area of memory to an output device, saving the data in binary format.

■ ■ ■

filespec identifies the file to be loaded, and has the form shown under "File and Device Information" in Section 3.

offset is an integer in the range 0 to 65535. This is the location at which saving is to start, and is the offset into the segment declared by the last DEF SEG statement.

length is an integer in the range 1 to 65535. It designates the length in bytes of the memory image to be saved.

filespec, offset, and **length** are required for this command to be executed. If **offset** and/or **length** are omitted, a "Bad file name" error message appears, and the save is terminated.

If the device name is omitted from **filespec**, the DOS default drive is assumed.

A DEF SEG statement should be executed before the BSAVE statement, since the address given in the last known DEF SEG statement is used for the save.

BLOAD and BSAVE are often used for loading and saving assembly language programs. (See CALL statement for how to execute assembly language programs from within a GWBASIC program.) However, any segment may be specified as the source or target for these statements with the DEF SEG statement. This allows you to save and display screen images by saving from or loading to the screen buffer.

```
10 'Save the graphic screen buffer
20 SCREEN 105                  'set screen attributes
                                for text and graphics
30 SCREEN ,,3,3                'set active and visual
                                pages to page 3.
```

(program generates a screen image)

```
100 DEF SEG = &H1800           'set segment pointer
                                to screen buffer
110 BSAVE "PICTURE",0,27000     'save screen buffer
                                 to file "PICTURE"
```

DEF SEG is used to set up the segment address of the screen buffer as hex 1800, which corresponds to the statement SCREEN ,,3,3. The offset of 0 and length 27000 specifies that the entire screen is to be saved.

NOTE: For 400-line systems, line 110 of this example should read:

```
110 BSAVE "PICTURE",0,33000
```

CALL Statement

FORMAT CALL **numvar[(variable[,variable]...)]**

PURPOSE Calls an assembly language subroutine.

■ ■ ■

numvar is the name of a numeric variable containing the starting point in memory of the subroutine being called. This starting point is specified as an offset into the current segment, which must have been previously defined in a DEF SEG statement.

variable is the name of a variable, or variables separated by commas, that is to be passed to the subroutine. Constants cannot be used.

```
100 DEF SEG=&H8000
110 FOO=0
120 CALL FOO(A,B$,C)
 .
 .
 .
```

Line 100 sets the segment address to 8000 hex. FOO is set to zero, which causes the call to FOO to execute the subroutine at location hex 80000. Variables A, B$, and C are passed as arguments to the assembly language subroutine.

CALLS Statement

FORMAT CALLS **numvar[(variable[,variable]...)]**

PURPOSE Calls an assembly language subroutine, passing segmented addresses of all arguments.

■ ■ ■

numvar is the name of a numeric variable containing the starting point in memory of the subroutine being called. This starting point is specified as an offset into the current segment, which must have been previously defined in a DEF SEG statement.

variable is the name of a variable, or variables separated by commas, that is to be passed to the subroutine. Constants cannot be used.

This statement is just like CALL except that the segmented addresses of all arguments are passed, while CALL passes unsegmented addresses. CALLS should be used when accessing routines written with the FORTRAN calling convention; all FORTRAN parameters are call-by-reference segmented addresses.

See the example given in the CALL statement.

CDBL Function

FORMAT v = CDBL(x)

PURPOSE Converts x to a double-precision number.

■ ■ ■

x may be any numeric expression.

See "How GWBASIC Converts Numbers from One Precision to Another" in Section 5 for rules on how to convert from one precision to another. See CINT and CSNG to convert numbers to integer and single precision.

```
10 A=454.67
20 PRINT A;CDBL(A)
RUN
 454.67   454.6700134277344
Ok
```

The value of CDBL(A) is accurate only to the second decimal place after rounding because the original value of A has only two decimal places.

CHAIN Statement

FORMAT CHAIN [MERGE]**filespec**[,[**line**][,ALL][,DELETE **range**]]

PURPOSE Calls a program and passes variables to it.

■ ■ ■

filespec identifies the program that is called, and has the form shown under "File and Device Information" in Section 3. For example:

```
CHAIN "A:PROG1"
```

line is either a line number, or an expression that evaluates to a line number, in the called program. It specifies the line at which the called program is to begin running. If it is omitted, execution begins at the first line in the called program.

```
CHAIN "A:PROG1",1000
```

Here line 1000 is not affected by a RENUM command. If a called program is renumbered, the CHAIN statement must be changed to point to the new line number.

ALL specifies that every variable in the current program is to be passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables to be passed (see COMMON statement). For example:

```
CHAIN "A:PROG1",1000,ALL
```

MERGE brings a section of code into the GWBASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file (i.e., it must have been stored using the SAVE command with the A option) if it is to be merged.

```
CHAIN MERGE "A:OVRLAY",1000
```

DELETE allows you to delete an overlay when it has been used, so that a new overlay may be brought in. **range** has the format:

```
[line number][-line number]
```

and the line numbers specified must exist, or an "Illegal function call" error occurs. An example of this option is:

```
CHAIN MERGE "A:OVRLAY2",1000,DELETE 1000-5000
```

This example deletes lines 1000 through 5000 of the current program before loading in the overlay (the called program).

The line numbers in **range** are affected by the RENUM command, unlike the value of **line** above.

NOTES:

1. The CHAIN statement leaves files open.

2. The CHAIN statement with MERGE option preserves the current OPTION BASE setting.

3. Omitting the MERGE option causes (1) the loss of the OPTION BASE setting in the called program, and (2) the loss of variable types or user-defined functions for use by the called program. Thus, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the called program.

4. When using the MERGE option, userdefined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, these functions will be undefined after the merge is complete.

5. CHAIN performs a RESTORE before running the chained program. Thus DATA statements in the chained program will be read from the beginning.

Example 1

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING USING
   COMMON TO PASS VARIABLES.
20 REM SAVE THIS MODULE ON DISK AS "PROG1"
   USING THE A OPTION.
30 DIM A$(2),B$(2)
40 COMMON A$(),B$()
50 A$(1)="VARIABLES IN COMMON MUST BE ASSIGNED"
60 A$(2)="VALUES BEFORE CHAINING."
70 B$(1)="": B$(2) =""
80 CHAIN "PROG2"
90 PRINT: PRINT B$(1): PRINT: PRINT B$(2):PRINT
100 END
```

```
10 REM THE STATEMENT "DIM A$(2),B$(2)" MAY
   ONLY BE EXECUTED ONCE.
20 REM HENCE, IT DOES NOT APPEAR IN THIS MODULE.
30 REM SAVE THIS MODULE ON THE DISK AS "PROG2"
   USING THE 'A' OPTION.
40 COMMON A$(),B$()
50 PRINT: PRINT A$(1);A$(2)
60 B$(1)="NOTE HOW THE OPTION OF SPECIFYING A
   STARTING LINE NUMBER"
70 B$(2)="WHEN CHAINING AVOIDS THE DIMENSION
   STATEMENT IN 'PROG1'."
80 CHAIN "PROG1",90
90 END
```

In the above, "PROG1" shows how to chain to another program named "PROG2". "PROG2" shows how to chain back to line 90 in "PROG1".

Example 2

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING USING
   THE MERGE AND ALL OPTIONS.
20 REM SAVE THIS MODULE ON THE DISK AS "MAINPRG".
30 A$="MAINPRG"
40 CHAIN MERGE "OVRLAY1",1010,ALL
50 END

1000 REM SAVE THIS MODULE ON THE DISK AS
     "OVRLAY1" USING THE A OPTION.
1010 PRINT A$; " HAS CHAINED TO OVRLAY1."
1020 A$="OVRLAY1"
1030 B$="OVRLAY2"
1040 CHAIN MERGE "OVRLAY2",1010,ALL,
     DELETE 10001050
1050 END

1000 REM SAVE THIS MODULE ON THE DISK AS
     "OVRLAY2" USING THE A OPTION.
1010 PRINT A$; " HAS CHAINED TO ";B$;"."
1020 END
```

The above shows how to use the MERGE option while chaining.

CHDIR Command

FORMAT CHDIR **pathname**

PURPOSE Changes the current directory.

■ ■ ■

pathname specifies the name of the directory which is to be the current directory and is a string of up to 63 characters, which must be enclosed in quotes. CHDIR works exactly like the DOS command CHDIR - see the DOS manual.

```
CHDIR "SALES"
```

makes SALES the current directory.

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

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

See also the MKDIR and RMDIR commands.

CHR$ Function

FORMAT v = CHR$(a)

PURPOSE Returns the character for a given ASCII code.

■ ■ ■

a must be in the range 0 to 255.

ASCII codes are listed in Appendix C.

CHR$ is commonly used to send a special character to the screen or printer. For instance, the BEL character, which beeps the speaker, might be included as CHR$(7) as a preface to an error message (instead of using BEEP). CHR$(12) might be a form feed to clear the screen and return the cursor to the home position.

Refer to the ASC function to see how to return the ASCII code for a character.

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

The character for ASCII code 66 is B.

```
10 IF INKEY$ <> CHR$(3) THEN 10
```

The program will loop at statement 10 until the character ^C (Ctrl-C) is entered from the keyboard.

CINT Function

FORMAT v = CINT(x)

PURPOSE Converts x to an integer.

■ ■ ■

x may be any numeric expression in the range -32768 to 32767.

x is converted to an integer by rounding the fractional portion. If x is not in the required range, an "Overflow" error occurs.

See FIX and INT, which also return integers. See also CSNG and CDBL for converting numbers to single or double precision.

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

Observe how rounding occurs.

CIRCLE Statement

FORMAT CIRCLE [STEP] **(x,y),radius [,color [,start,end[,aspect]]]**

PURPOSE Draws a circle, arc or ellipse with the specified center and radius.

■ ■ ■

This statement can only be used in graphics mode (SCREEN 1, 2, 104 or 105).

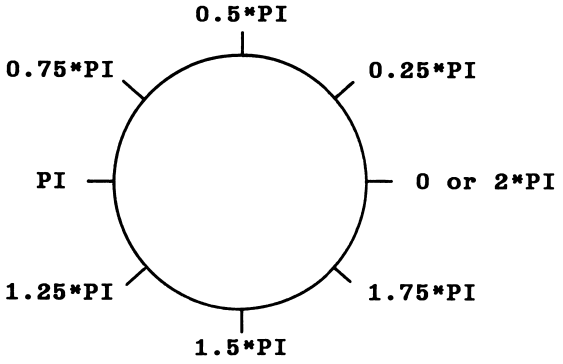
x is the x-coordinate for the center of the circle.

y is the y-coordinate for the center of the circle.

radius is the radius of the circle in pixels.

color is the number of the color in which the figure is to be drawn. In medium resolution (SCREEN 1), **color** ranges from 0 to 3, where 0 indicates the background color and 1 to 3 denote colors from the current palette (see Table 7-2). In high resolution (SCREEN 2) and super resolution (SCREEN 104 and 105), **color** can be either 0 (background color) or 1 (foreground color). Default is 3 for medium resolution and 1 for high- and super resolution. Default for monochrome screens is the foreground color.

start and **end** are angles, allowing you to specify where drawing of the figure will begin and end. The range is -2*pi through 2*pi; see the diagram below. If the start and end angles are negative, the figure will be connected to the center point with a line, and the angles will be treated as if they were positive. Note that this is different from adding 2*pi. **start** may be less than **end**.



NOTE: The CIRCLE statement always draws counterclockwise from start to end.

If you specify coordinates outside the screen or current viewport, this is not regarded by GWBASIC as an error. The overlapping portion of the figure will simply not be drawn.

aspect is the aspect ratio, i.e., the ratio of the x radius to the y radius. The default ratio produces a round circle on the screen.

If the aspect ratio is less than one, the radius given is the x radius. If it is greater than one, the y radius is given.

The last point referenced after a circle is drawn is the center of the circle.

Coordinates can be shown as absolutes, as in the syntax shown above, or the STEP option can be used to reference a point relative to the most recent point used. The format of the STEP option is:

STEP (x,y)

For example, if the most recent point referenced were (x,y), STEP (10,5) would reference the point (x+10,y+5).

Assume that the last point plotted was 100,50. Then

```
CIRCLE (200,200),50
```

and

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

will both draw a circle at 200,200 with radius 50. The first example uses absolute notation; the second uses relative notation.

```
10 CLS:SCREEN 105:PI = 3.1415926#
20 CIRCLE (320,160),30,,-1.1*PI,-0.9*PI
30 CIRCLE (310,150),3
```

The above example illustrates how specifying a negative start and end value causes the figure to be connected to the center point with a line. The program draws the face of a well-known arcade game character.

```
10 'These examples demonstrate use of the
    CIRCLE statement
20 SCREEN 105: CLS: PI=3.1415926#
30 CIRCLE (50,100),40,,0,2*PI
40 CIRCLE (150,100),40,,,,1.5
50 CIRCLE (250,100),40,,0.25*PI,1.5*PI
60 CIRCLE (350,100),40,,-0.25*PI,1.5*PI
70 CIRCLE (450,100),40,,0.25*PI,-1.5*PI
80 CIRCLE (550,100),40,,1.25*PI,-0.75*PI
```

Examining these examples individually will help you further understand operation of the CIRCLE statement.

CLEAR Command

FORMAT CLEAR [,[**expression1**][,**expression2**]]

PURPOSE Sets all numeric variables to zero, all string variables to null, and closes all open files. Options set the end of memory and the amount of stack space available to GWBASIC.

■ ■ ■

expression1 is a memory location which, if specified, is the highest location available for the GWBASIC work space (where your program and data are stored, along with the interpreter work area). This expression is useful, for example, in reserving space for assembly language programs. Note that the maximum value of **expression1** is 65534.

expression2 sets aside stack space for GWBASIC. The default is 768 bytes or one-eighth of the available memory, whichever is smaller. This expression is useful, for example, if there are nested GOSUB statements or FOR...NEXT loops in your program, or if you use PAINT to do complex scenes.

Using CLEAR

- closes all files
- clears all COMMON variables
- resets numeric variables and arrays to zero
- resets the stack and string space
- resets string variables and arrays to null
- releases all disk buffers
- resets all DEF FN and other DEF statements

CLEAR resets SOUND to music foreground and PEN and STRIG to OFF.

To free memory without erasing all your data, use the ERASE statement.

```
CLEAR
```

The above clears all data from memory without erasing the program.

```
CLEAR ,32768
```

The above clears the data and sets the maximum workspace size to 32K.

```
CLEAR ,,2000
```

The above clears the data and sets the size of the stack to 2000 bytes.

```
CLEAR ,32768,2000
```

The above clears data, sets the maximum work space for GWBASIC to 32K, and sets the stack size to 2000 bytes.

CLOSE Statement

FORMAT CLOSE **[[#]filenum[,[#]filenum...]]**

PURPOSE Concludes I/O to a disk file or device.

■ ■ ■

filenum is the number used to OPEN the file.

CLOSE with no file numbers closes all open devices and files.

CLOSE ends the association between a particular file or device and its file number that was created by an OPEN statement. The file or device may be opened again using the same or a different number, or the number may be reused to open any device or file.

If a file or device has been opened for sequential output, CLOSE causes the final buffer to be written to it.

Executing SYSTEM, CLEAR, RESET, END, NEW, and RUN automatically closes all open files and devices. STOP does not close any files or devices.

```
10 CLOSE 2,5
```

or

```
10 CLOSE #2,#5
```

Either statement closes files 2 and 5.

CLS Statement

FORMAT CLS

PURPOSE Erases contents of entire current screen and homes the cursor

■ ■ ■

In text mode CLS clears the active page (see the SCREEN statement) to the background color (see the COLOR statement) and homes the cursor to the upper left corner.

In graphics mode (whether super-, medium-, or high resolution) CLS clears the screen buffer to the background color and homes the cursor to the upper left corner of the screen.

Entering Ctrl-Home from the keyboard is equivalent to executing a CLS statement.

If a viewport has been made active by means of the VIEW statement, a subsequent CLS statement only clears the viewport. If you want to clear the whole screen, use VIEW with no parameters to make the entire screen the current viewport, and then execute CLS.

```
10 CLS
20 PRINT "Hi there!!!"
```

The above example clears the screen, and types out the message "Hi there!!!" in the upper left corner of the screen.

COLOR Statement (Graphics)

FORMAT COLOR **[background][,palette]**

PURPOSE Sets background color and selects a color palette for the foreground.

■ ■ ■

This form of the COLOR statement may be used with most compatible color/graphics monitor adapters in medium resolution (SCREEN 1). Attempts to use COLOR in high-resolution mode (SCREEN 2) will result in an "Illegal function call" error. In super-resolution mode, the COLOR statement gives an "Illegal function call" error for SCREEN 104 and is ignored for SCREEN 105.

Each character displayed on the screen is made up of a background and a foreground color. The foreground is the color of the character itself, and the background is the color of the space around the character. If you are using color graphics, you use this form of the COLOR statement to set the foreground and background colors, then you display the characters by means of statements such as PRINT, LINE, PAINT etc.

If you are using color in graphics mode (see the SCREEN statement), each COLOR statement with the format shown below allows you to choose one of 16 colors for the background and one of two "palettes" of three colors each for the foreground. A palette allows you to select colors in subsequent LINE, PAINT etc. statements without having to specify a new color statement each time.

background is a number in the range 0 to 15 that specifies a background color as shown in Table 7-1 below.

palette can be any unsigned integer or numeric expression. If its value is odd, palette 1 is selected, otherwise palette 0 is selected. Table 7-2 specifies the effective colors with each palette. A value outside the range 0-255 will cause an "Illegal function call" error.

Table 7-1

COLOR NUMBERS

No.	Color	No.	Color
0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-Intensity White

Table 7-2

PALETTE INFORMATION

Color	Palette 0	Palette 1
1	Green	Cyan
2	Red	Magenta
3	Brown	White

Foreground color may be the same as the background color, thus making displayed characters invisible.

Any parameter may be omitted. If a parameter is omitted, the previous value is retained.

Colors selected by the COLOR statement will be used by the PSET, PRESET, LINE, CIRCLE, PAINT, and DRAW statements.

Values entered outside the ranges will result in an "Illegal function call" error, and previous values will be retained.

The following examples assume that you have a color/ graphics monitor adapter present.

```
10 SCREEN 1
20 COLOR 9,0
```

Sets background to light blue, selects palette 0.

```
30 COLOR ,3
```

Background stays light blue, selects palette 1 because palette is an odd number.

```
10 CLS
20 SCREEN 1,0          'Select medium resolution -
                        color ON
30 COLOR 10,1          'Select light green
                        background - palette 1
40 PRINT "Characters are white (color 3)"
50 LINE (100,100)-(200,150),2,B  'Draw a box with
                                  magenta lines
60 PAINT (150,125),1,2  'Fill the box with cyan
```

With a standard monochrome system this program will print in reverse video, and draw and fill the box in white.

After running this program, pressing the F10 key returns the screen to the normal mode.

COLOR Statement (Text)

FORMAT COLOR **[foreground][,[background][,border]]**

PURPOSE Sets the foreground (i.e., character) color, background color, and border color respectively.

■ ■ ■

This form of the COLOR statement can only be used in text mode (SCREEN 0 or 100). See the COLOR Statement (Graphics) for an explanation of foreground and background colors.

foreground is a number in the range 0 to 31 that specifies a color as in Table 7-1, which is repeated below. Values 16 to 31 are the same colors as for 0 to 15 but the characters are set blinking.

background and **border** are numbers in the range 0 to 15 and also specify colors according to Table 7-1. They set the color of the text background and the border round the edge of the screen. Note that **border** has no effect with a monochrome screen.

Table 7-1

COLOR NUMBERS

No.	Color	No.	Color
0 (16)	Black	8 (24)	Gray
1 (17)	Blue	9 (25)	Light Blue
2 (18)	Green	10 (26)	Light Green
3 (19)	Cyan	11 (27)	Light Cyan
4 (20)	Red	12 (28)	Light Red
5 (21)	Magenta	13 (29)	Light Magenta
6 (22)	Brown	14 (30)	Yellow
7 (23)	White	15 (31)	High-Intensity White

For foreground colors, values in parentheses cause the character to blink.

The effect of each of these parameters varies depending on the monitor and monitor adapter used (see Tables 7-3 and 7-4). Values outside the ranges shown will cause an "Illegal function call" error.

Table 7-3

COLOR ON A STANDARD MONOCHROME MONITOR

Foreground (0-31)	Background (0-15)	
	Even	Odd
0	Black on black	Black on white
1	White on black underlined	Black on white underlined
2-7	White on black	Black on white
8-15	Same as 0-7 except high intensity.	
16-23	Same as 0-7 except blinking.	
24-31	Same as 0-7 except blinking and high intensity.	

NOTE: Border has no effect with a monochrome screen and may be omitted. If it is supplied, it must be in the range 0-15 or an "Illegal function call" error will result.

The following example demonstrates use of the COLOR statement in text mode on a standard monochrome system.

```
10 CLS
20 SCREEN 100          'SCREEN 0,0 could also be used
30 COLOR 7,0: PRINT "Normal Mode - white on black"
40 COLOR 1,0: PRINT "Normal Mode - underlined"
50 COLOR 15,0: PRINT "Normal Mode - high intensity"
60 COLOR 23,0: PRINT "Normal Mode - blinking"
70 COLOR 0,7: PRINT "Reverse Video - black on white"
80 COLOR 24,7: PRINT "Reverse Video - high intensity
   & blinking"
90 COLOR 7,0           'Back to normal mode
```

Table 7-4

COLOR IN TEXT MODE WITH A COLOR/GRAPHICS ADAPTER

Range	Foreground (0-31)	Background (0-15)	Border (0-15)
0-7	colors 0-7	colors 0-7	colors 0-7
8-15	colors 8-15	colors 0-7	colors 8-15
16-31	colors 0-15 blinking	error	error

The following example demonstrates the use of COLOR in text mode with a color/graphics adapter.

```
10 CLS
20 SCREEN 101          'SCREEN 0,1 could also be
                        used
30 COLOR 7,0: PRINT "Normal - white on black"
40 COLOR 28,1: PRINT "Blinking - light red on
   blue"
50 COLOR 0,7: PRINT "Reverse Video - black on
   white"
60 COLOR 7,0           'Return to normal
```

COM(n) Statement

FORMAT COM(n) ON

COM(n) OFF

COM(n) STOP

PURPOSE Enables or disables event trapping of communications activity on the specified channel.

n is the number of the communications channel: 1, 2, 3 or 4.

The COM(n) ON statement enables communications event trapping by an ON COM statement. While trapping is enabled, and if a nonzero line number is specified in the ON COM statement, GWBASIC checks between every statement to see if any characters have been received on the specified channel. If so, the ON COM statement is executed to pass control to a subroutine that can perform any desired processing.

COM(n) OFF disables communications event trapping. If characters are received, this event is not remembered by GWBASIC.

COM(n) STOP disables communications event trapping, but if an event occurs, it is remembered and ON COM will be executed as soon as trapping is enabled.

```
10 COM(1) ON
```

Enables error trapping of communications activity on channel 1 (COM1:).

COMMON Statement

FORMAT COMMON **list of variables**

PURPOSE Passes variables to a chained program.

■ ■ ■

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Some software products allow the number of dimensions in the array to be included in the COMMON statement. GWBASIC will accept that syntax, but will ignore the numeric expression itself. For example, the following statements are both valid and are considered equivalent:

```
COMMON A(1)
COMMON A(3)
```

The number in parentheses is the number of dimensions, not the dimensions themselves. For example, the variable A(3) in this example might correspond to a DIM statement of DIM A(5,8,4).

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

The above chains the program to PROG3. Variables A,B,C and G$ as well as the array D are passed.

CONT Command

FORMAT CONT

PURPOSE Continues program execution after a break.

■ ■ ■

This command restarts a program after Ctrl-Break or Ctrl-C has been pressed, a STOP or END statement has been executed, or an error has occurred. Execution resumes at the point where the break occurred. The prompt symbol is redisplayed if the break occurred while the program was waiting for input from the keyboard.

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, you can examine or change the values of variables using direct mode statements. Use CONT to resume program execution, or use the direct mode GOTO, which requires a specific line number to resume.

CONT is invalid if the program has been edited during the break.

COS Function

FORMAT v = COS(x)

PURPOSE Returns the cosine function of an angle.

■ ■ ■

x must be in radians. To convert from degrees to radians, multiply the degrees by PI/180, where PI=3.141593.

The calculation of COS(x) is performed in single precision, unless the the /D switch was specified in the GWBASIC command. In this case the calculation will be performed in double precision, provided that either the variable receiving the cosine value is of double-precision type, or x is specified as double-precision using the # sign.

```
10 X=2*COS(0.4)
20 PRINT X
RUN
 1.842122
Ok
```

The above shows the result of two times the cosine of 0.4.

CSNG Function

FORMAT v = CSNG(x)

PURPOSE Converts x to a single-precision number.

■ ■ ■

x may be any numeric expression. CINT and CDBL functions convert numbers to integers and double-precision numbers.

```
10 A# = 975.3421115#
20 PRINT A#; CSNG(A#)
RUN
 975.3421115   975.3421
Ok
```

The value of the double-precision number A# is returned as CSNG(A#).

CSRLIN Function

FORMAT v = CSRLIN

PURPOSE Returns the current line position of the cursor.

■ ■ ■

The value returned will be in the range 1 to 24.

For the column position of the cursor, see POS function.

LOCATE moves the cursor to a specified position.

```
10 Y = CSRLIN
20 X = POS(0)
30 LOCATE 20,1 :PRINT "HELLO"
40 LOCATE X,Y
```

Line 10 records the current line, line 20 the current column. Line 30 causes HELLO to be printed on line 20. Last, in line 40 the cursor is returned to its original position.

CVI, CVS, CVD Functions

FORMAT `v = CVI(2-byte string)`
`v = CVS(4-byte string)`
`v = CVD(8-byte string)`

PURPOSE Converts string variables to numeric variables.

■ ■ ■

Numeric values read from a random file on disk must be converted from strings back into numbers. CVI converts a two-byte string to an integer. CVS converts a four-byte string to a single-precision number. CVD converts an eight-byte string to a double-precision number.

To convert numeric values to string values, see MKI$, MKS$, MKD$ functions.

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

Line 70 has random file (#1) with certain fields. Line 80 reads a record from the file, and line 90 uses CVS function to interpret (N$) as a single-precision number.

DATA Statement

FORMAT DATA **list of constants**

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

■ ■ ■

DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line. These must be separated by commas. Any number of DATA statements may be used in a program. READ statements access DATA statements in line number order. The information given in these statements may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

list of constants may contain numeric constants in any format whether fixed-point, floating-point, or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

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

Use RESTORE to reread DATA statements from the beginning.

See the READ statement for an example.

DATE$ Statement

FORMAT DATE$ = **string expression**

PURPOSE Sets the date.

■ ■ ■

string expression has one of the following forms:

```
mm-dd-yy
mm/dd/yy
mm-dd-yyyy
mm/dd/yyyy
```

```
10 DATE$="02/11/85"
```

The current date is set to February 11, 1985.

DATE$ Variable

FORMAT **v$ = DATE$**

PURPOSE Returns the system date.

■ ■ ■

This variable returns a ten-character string in the form mm-dd-yyyy, where mm is the month (01 through 12), dd is the day (01 through 31), and yyyy is the year (1980 through 2099).

```
10 PRINT DATE$
```

The date will be printed as set with the DATE$ statement.

DEF FN Statement

FORMAT DEF FN**name[(parameter list)]=function definition**

PURPOSE Defines and names a function written by the user.

■ ■ ■

name must be a legal variable name. This name, preceded by FN, becomes the name of the function.

parameter list is optional and is used if variable names in the function definition are to be replaced with values when the function is called. The list consists of the appropriate variable names, separated by commas.

function definition is an expression that performs the operation of the function. It is limited to one logical line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

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

This statement may define either numeric or string functions. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be encountered before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in direct mode.

```
10 DEF FNFUN (X) = N^2
20 INPUT "Number";N
30 PRINT "Square is" FNFUN(N)
RUN
 Number? 5
 Square is 25
```

Line 10 defines the function FUN. The function is called in line 30.

DEF SEG Statement

FORMAT DEF SEG [=**address**]

PURPOSE Assigns the current segment address to be referenced by a subsequent BLOAD, BSAVE, CALL, CALLS, or POKE statement or by a USR or PEEK function.

■ ■ ■

address is an integer in the range 0 to 65535.

The address specified is saved for use as the segment required by BLOAD, BSAVE, CALL, CALLS, POKE, USR, and PEEK.

Entering values outside the address range will result in an "Illegal function call" error, and the previous value will be retained.

If **address** is omitted, the segment to be used is set to the GWBASIC data segment (DS). This is the initial default value.

If **address** is given, it should be based on a 16-byte boundary. GWBASIC does not check the validity of the specified address.

NOTE: DEF and SEG must be separated by a space. Otherwise, GWBASIC will interpret the statement DEFSEG=100 to mean "assign the value 100 to the variable DEFSEG".

```
10 DEF SEG=&HB800        'Set segment to B800 Hex
20 DEF SEG               'Restore segment to
                          GWBASIC data segment
```

The above lines set and restore data segments.

DEFtype Statements

FORMAT DEF**type range(s) of letters**

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

■ ■ ■

type is INT, SNG, DBL, or STR.

Any variable names beginning with the letter(s) specified in **range of letters** will be considered the type of variable specified in the **type** portion of the statement. However, a type declaration character (%, !, # or $) always takes precedence over a DEFtype statement.

If no type declaration statements are encountered, GWBASIC assumes that all variables without declaration characters are single-precision variables.

```
10 DEFDBL L-P
```

All variables beginning with the letters L, M, N, O, and P will be double-precision variables.

```
10 DEFSTR A
```

All variables beginning with the letter A will be string variables.

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

All variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

DEF USR Statement

FORMAT DEF USR[**digit**]=**integer expression**

PURPOSE Specifies the starting address of an assembly language subroutine.

■ ■ ■

digit may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose **address** is being specified. If **digit** is omitted, DEF USR0 is assumed.

integer expression is the starting address of the USR routine.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

```
.
.
.
200 DEF USR0=24000
210 X=USR0(Y^2/2.89)
.
.
.
```

This example shows a call to a subroutine at absolute location 24000.

DELETE Command

FORMAT DELETE **[line number][-][line number]**

PURPOSE Deletes program lines.

■ ■ ■

GWBASIC always returns to command level after a DELETE is executed.

If **line number** does not exist, an "Illegal function call" error occurs.

DELETE 40

Deletes line 40.

DELETE 40-100

Deletes lines 40 through 100, inclusive.

DELETE -40

Deletes all lines up to and including line 40.

DELETE 40-

Deletes lines 40 to the end, inclusive.

DIM Statement

FORMAT DIM **list of subscripted variables**

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

■ ■ ■

If an array variable name is used without a DIM statement, the maximum value of the array's subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

The DIM statement sets all the elements of the specified numeric arrays to an initial value of zero, and the elements of string arrays to null.

Theoretically, the maximum number of dimensions allowed in a DIM statement is 255. In reality, however, that number would be impossible, since the name and punctuation are also counted as spaces on the line, and the line itself has a limit of 255 characters. The number of dimensions is further limited by the amount of available memory.

If the default dimension (10) has already been established for an array variable and that variable is later encountered in a DIM statement, the message "Array already dimensioned" is displayed. Thus it is good programming practice to put the required DIM statements at the beginning of a program, outside any processing loops.

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

Reads twenty numbers from DATA statements into array A.

DRAW Statement

FORMAT DRAW **string expression**

PURPOSE Draws a line as indicated by **string expression.**

■ ■ ■

This statement can only be used in graphics mode (SCREEN 1,2, 104 or 105).

string expression is one or more subcommands that call for motion (up, down, left, right), color, angle, and scale factor. Subcommands in the string can be separated by optional spaces or semicolons.

Each of the following subcommands begins movement from the "current graphics position". This is usually the coordinate of the last graphics point plotted with LINE or PSET. The default is the center of the screen.

If no argument is supplied, the commands listed below move one unit. The size of a unit can be modified by the S subcommand, which sets the scale factor (the default unit size is one point). In all the following commands, **n** is a numeric argument which can be a constant like "123", or "**=variable**", in which the name of a variable is specified.

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
F **n**	Move diagonally up and left
G **n**	Move diagonally down and left
H **n**	Move diagonally down and right
M **x,y**	Move absolute or relative. If **x** is preceded by a plus (+) or minus (-), **x** and **y** are added to the current graphics position and connected with the current position by a line. Otherwise, a line is drawn to point **x,y** from the current cursor position.

The following prefix commands may precede any of the above movement commands:

B Move but do not plot any points.

N Move but return to original position when done.

A **n** Set angle of rotation **n**. **n** may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270. Figures rotated 90 or 270 degrees are scaled so they will appear the same size as 0 or 180 degrees on a monitor screen with the standard aspect ratio of 4/3.

C **n** Set color **n** in which line is to be drawn. In medium resolution (SCREEN 1), **n** ranges from 0 to 3, where 0 indicates the background color and 1 to 3 denote colors from the current palette (see Table 7-2). In high resolution (SCREEN 2) and super resolution (SCREEN 104 and 105), **n** can be either 0 (background color) or 1 (foreground color). Default is 3 for medium resolution and 1 for high- and super resolution. Default for monochrome screens is the foreground color.

S **n** Set scale factor. **n** may range from 1 to 255. The scale factor multiplied by the distances given with U, D, L, R, or relative M commands gives the actual distance traveled.

X **string expression;**
Execute substring. This command allows you to execute a substring from a string. You can have one string execute another, which executes a third, and so on. A semicolon (;) is mandatory at the end of an X command.

TA **n** Rotate figure **n** degrees. **n** must be in the range -360 to 360 degrees. If **n** is positive, rotation is counter-clockwise; if negative, rotation is clockwise.

P **x,y** Paint the object drawn using the specified colors: **x** is an integer defining the paint color, and **y** is an integer defining the border color.

```
10 SCREEN 105
20 CLS
30 PSET (160,100),1
40 DRAW "U20 R30 D20 L30"
```

The above will draw a box on the screen.

```
10 SCREEN 105:CLS
20 FOR D=0 TO 360
30 DRAW "TA=D; NU100"
40 NEXT D
```

This example draws spokes radiating from a central point, and illustrates the use of TA for angle rotation.

EDIT Command

FORMAT EDIT **line number**

PURPOSE Enters edit mode at the specified line.

■ ■ ■

After you enter the line number to be edited, GWBASIC displays the entire line ready for editing, placing the cursor on the first character of the line.

To edit the current line, use a period (.) instead of **line number**. This is useful if you have just edited a line and pressed <Return> but decide you want to edit the line again, or if you want to edit the last line of a program that has just executed.

END Statement

FORMAT END

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

■ ■ ■

END statements may be placed anywhere in the program to terminate execution. Unlike STOP, END does not cause a "Break" message to be printed, and END closes all files.

An END statement at the end of a program is optional. GWBASIC always returns to command level after an END is executed.

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

The program ends if K is greater than 1000. If it is less than or equal to 1000, it branches to line 20.

ENVIRON Statement

FORMAT ENVIRON **param** [=] **string**

PURPOSE Adds or modifies a parameter in the BASIC environment table, allowing you to change the table's PATH parameter for a child process, or to pass new parameters to a child process.

■ ■ ■

param is the name of the parameter to be added or modified.

string is the new parameter value.

param and **string** must be separated by either an equal sign (=) or a space. Everything to the left of the equal sign or space will be assumed to be a parameter, and everything to the right, text.

If **param** does not already exist, it will be appended to the end of the table. If it does already exist, the old parameter is deleted and the new one appended to the end of the table.

If **string** is a null string (""), or consists only of a semicolon (";") the existing parameter is removed from the table, and the remainder of the table compressed.

An error message will be displayed if you specify parameters that are not strings, and "Out of memory" will occur when no more space can be allocated to the environment table. The amount of free space in the table is usually quite small.

The following example creates a default path to the root directory on drive A:

```
ENVIRON "PATH=A:\"
```

At this point, if GWBASIC executes a SHELL command with no parameters, the system displays the A> prompt (if A: is the current drive), allowing you to execute one or more DOS commands from the COMMAND.COM file. You could now, for example, execute the DOS command CHDIR to access a different directory and run a program from that directory under DEBUG, using the path specified in the ENVIRON statement to access DEBUG from the root directory. When the program ended, you would type EXIT to return to the original BASIC program.

You can add a new parameter to the environment table as follows:

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

Assuming the path specified earlier, the environment table will now contain this string (note that you can read the contents of the table by using the ENVIRON$ function):

```
PATH=A:\;SESAME=PLAN
```

If you then give a new value for the PATH parameter, like this:

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

the environment table will now contain:

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

If you want to add data to the end of the PATH parameter, you can use the ENVIRON$ function with this statement to avoid entering the entire PATH parameter over again:

```
ENVIRON "PATH="+ENVIRON$("PATH")+"B:\PLAN"
```

At this point the table will contain:

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

Finally, you can delete the parameter SESAME:

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

leaving the following in the table:

```
PATH=A:\SALES;A:\ACCOUNTING;B:\PLAN
```

See also the ENVIRON$ function and the SHELL command.

ENVIRON$ Function

FORMAT `v$ = ENVIRON$ (param)`
`v$ = ENVIRON$ (n)`

PURPOSE Returns a parameter value from the BASIC environment table.

■ ■ ■

param is a string containing the name of the parameter for which the value is to be returned.

n is an integer expression returning a value in the range 1 to 255.

If you use **param**, ENVIRON$ returns a string containing the text following "**param** = " in the environment table. This string must not exceed 255 characters, otherwise a "String too long" message is displayed. If the parameter cannot be found, or if it has no text following it, a null string is returned.

If you use **n**, the nth parameter in the table is returned, together with its value. If the nth parameter does not exist, a null string is returned.

As an example, if the environment table contains:

```
PATH=A:\SALES;A:\ACCOUNTING;B:\PLAN
```

the statement:

```
PRINT ENVIRON$("PATH")
```

will print the string:

```
A:\SALES;A:\ACCOUNTING;B:\PLAN
```

The first two entries in the environment table when the system is booted are the PATH parameter (which initially has a null value) and a parameter named COMSPEC, which tells the system where to find the COMMAND.COM file.

Assuming the PATH parameter has been modified according to the example above, the statement:

```
PRINT ENVIRON$(1)
```

will print the string:

```
PATH=A:\SALES;A:\ACCOUNTING;B:\PLAN
```

Notice how this form of ENVIRON$ causes the parameter name to be printed as well.

The following program saves the BASIC environment table in an array so that you can modify it for a child process. After the child process finishes, the environment is restored.

```
10 DIM TBL$(10) 'assume no more than 10 params
20 PARMS = 1     'initial no. of params
30 WHILE LEN(ENVIRON$(PARMS)) > 0
40 TBL$(PARMS) = ENVIRON$(PARMS)
50 PARMS = PARMS + 1
60 WEND
70 PARMS = PARMS - 1 'adjust to correct number
80 'now store new environment
90 ENVIRON "MYCHILD.PARM1 = SORT BY NAME"
100 ENVIRON "MYCHILD.PARM2 = LIST BY NAME"
         .
         .
         .
1000 SHELL "MYCHILD" 'runs MYCHILD.EXE
1010 FOR X = 1 TO PARMS
1020 ENVIRON TBL$(X) 'restore params
1030 NEXT X
         .
         .
         .
```

See also the ENVIRON and SHELL statements.

EOF Function

FORMAT **v = EOF(file number)**

PURPOSE Tests for the end-of-file condition.

■ ■ ■

This function returns -1 (true) if the end of a sequential file has been reached. Use it while inputting to avoid "Input past end" errors.

When EOF is used with random access files, it returns -1 (true) if the last executed GET statement was unable to read an entire record because of an attempt to read beyond the end.

When EOF is used with a communications device, the definition of the end-of-file condition is dependent on the mode (ASCII or binary) in which the device was opened. In binary mode, EOF is true when the input queue is empty (LOC(n)=0). It becomes false when the input queue is not empty. In ASCII mode, EOF is false until a Ctrl-Z is received, and from then on it will remain true until the device is closed.

```
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
    .
    .
    .
```

Reads information from a file named "DATA" into the array M, and when the end of file is reached the program moves to line 100.

ERASE Statement

FORMAT ERASE **arrayname[,arrayname]...**

PURPOSE Deletes arrays from memory.

■ ■ ■

arrayname is the name of an array you want to erase.

After the arrays are erased, they may be redimensioned, or the free space may be used for other purposes. If you try to redimension an array without first erasing it, a "Duplicate definition" error occurs.

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

The above shows how to redimension array "B".

ERDEV and ERDEV$ Variables

FORMAT
v = ERDEV
v$ = ERDEV$

PURPOSE These two read-only variables return the last device error code issued (ERDEV) and the name of the device causing the error (ERDEV$).

■ ■ ■

ERDEV is an integer which, in its lower 8 bits contains the interrupt 24 error code returned by the last device to declare an error, and in its upper 8 bits contains the word attribute bits (13, 14 and 15) of the device header block.

If the error was on a character device, ERDEV$ contains the 8-byte name of the device driver. If the error was not on a character device, ERDEV$ contains the two-character block device name (A:, B:, C: etc.)

If a user-installed device driver, "MYLPT2", ran out of paper, and the driver's error number for that problem was "9":

```
PRINT ERDEV, ERDEV$
```

will yield:

```
9          MYLPT2
```

ERR and ERL Variables

FORMAT ERR

ERL

PURPOSE Returns the error code and line number where the error occurred.

■ ■ ■

When an error-handling routine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error handling routine. Refer to ON ERROR statement in this section.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test whether an error occurred in a direct statement use the form:

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

Otherwise, use

```
IF ERR = error code THEN ...

IF ERL = line number THEN ...
```

Be sure to put the line number on the right side of the relational operator so it can be renumbered by RENUM.

Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement.

GWBASIC error codes are listed in Appendix J. See also ERROR statement.

```
10 ON ERROR GOTO 60
20 OPEN "I",#1,"JUNK.DAT"
30 CLOSE #1
40 PRINT "The file exists"
50 END
60 IF ERR=53 AND ERL=20 THEN PRINT "File does
   not exist"
70 PRINT "Error";ERR;"at line";ERL
80 STOP
```

The above shows a way to check for the existence of a file.

ERROR Statement

FORMAT ERROR **n**

PURPOSE Simulates the occurrence of a GWBASIC error; or allows you to define your own error codes.

■ ■ ■

n must be an integer expression between 1 and 255.

If the value of **n** is the same as an error code already in use (see Appendix F), the ERROR statement will simulate the occurrence of that error and display the corresponding error message. (See Example 1.)

To define your own error code, use a value different from those used by GWBASIC. (We suggest you use the highest available values to allow for additional error codes being added to GWBASIC.) Your new error code may then be tested in an error-handling routine. (See Example 2.)

If you have defined an error-handling routine with ON ERROR, the program will enter this routine when it encounters the error.

If an ERROR statement specifies a code for which no error message has been defined, the message "Unprintable error" is displayed, and execution halts.

Example 1

```
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                (you type this line)
String too long         (GWBASIC displays this line)
Ok
```

The above shows how to simulate a "String too long" error.

Example 2

```
        .
        .
        .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B>5000 THEN ERROR 210
        .
        .
        .
400 IF ERR=210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL=130 THEN RESUME 120
        .
        .
        .
```

The above traps an error with code 210.

EXP Function

FORMAT `v = EXP(x)`

PURPOSE Returns the exponential function of the mathematical expression **e** (base of natural logarithms).

■ ■ ■

x may be any numeric expression but no greater than 88.02969.

If **x** is greater than 88.02969, an "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

EXP returns a single-precision value unless the /D switch was used with the GWBASIC command. In this case the calculation will be in double precision, provided that either the variable that receives the exponential function is of double-precision type, or **x** is specified as double-precision using the # sign.

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

This example calculates **e** raised to the power of 4.

FIELD Statement

FORMAT FIELD **[#]file number,field width** AS **string variable...**

PURPOSE Allocates space for variables in a random file buffer.

■ ■ ■

Before a GET statement or PUT statement can be executed, you must use FIELD to format the random file buffer.

file number is the number under which the file was opened.

field width is the number of characters to be allocated to **string variable.**

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

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed will remain in effect at the same time.

NOTE: Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is defined in a FIELD statement, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to the variables stored in string space.

```
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 place any data in the random file buffer.

```
10 OPEN "R,"#1,"A:PHONELST",35
15 FIELD #1,2 AS RECNBR$,33 AS DUMMY$
20 FIELD #1,25 AS NAME$,10 AS PHONENBR$
25 GET #1
30 TOTAL=CVI(RECNBR$)
35 FOR I=2 TO TOTAL
40 GET #1, I
45 PRINT NAME$, PHONENBR$
50 NEXT I
```

The above illustrates a multiply-defined FIELD statement. In statement 15, the 35-byte field is defined for the first record to keep track of the number of records in the file. In the next loop of statements (35-50), statement 20 defines the field for individual names and phone numbers.

The following example:

```
10 FOR LOOP%=0 TO 7
20 FIELD #1,(LOOP%*16) AS OFFSET$,16 AS
   A$(LOOP%)
30 NEXT LOOP%
```

shows the construction of a FIELD statement using an array of elements of equal size. The result is equivalent to the single declaration:

```
FIELD #1,16 AS A$(0),16 AS A$(1),...,16 AS
      A$(6),16 AS A$(7)
```

The next example:

```
10 DIM SIZE% (4%): REM ARRAY OF FIELD SIZES
20 FOR LOOP%=0 TO 4
30 READ SIZE% (LOOP%)
40 NEXT LOOP%
50 DATA 9,10,12,21,41
    .
    .
    .
120 DIM A$(4%): REM ARRAY OF FIELDED VARIABLES
130 OFFSET%=0
140 FOR LOOP%=0 TO 4%
150 FIELD #1,OFFSET% AS OFFSET$,SIZE%(LOOP%)
    AS A$(LOOP%)
160 OFFSET%=OFFSET%+SIZE%(LOOP%)
170 NEXT LOOP%
```

creates a field in the same manner as the previous example. However, the element size varies with each element. The equivalent declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$(1),
      ...SIZE%(4%) AS A$(4%)
```

FILES Statement

FORMAT FILES ["filespec"]

PURPOSE Displays the names of files on the specified disk.

■ ■ ■

filespec is the file specification (see Section 3, "File and Device Information") and includes a filename or pathname and optional device designation.

If **filespec** is omitted, all the files in the current directory on the current drive will be listed. **filespec** may contain question marks (?) or asterisks (*) used as wild cards. A question mark will match any single character in the filename or extension.

An asterisk will match one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks.

```
FILES
```

Shows all files in the current directory on the current drive.

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

Shows all files with extension ".BAS".

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

Shows all files on drive B.

```
FILES "B:"
```

This is equivalent to "B:*.*".

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

Shows all five-letter files whose names start with "TEST" and end with the .BAS extension.

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

Shows a subdirectory or file in the current directory.

FIX Function

FORMAT v = FIX(x)

PURPOSE Truncates x to an integer.

■ ■ ■

x may be a numeric expression.

FIX returns the value of the digits to the left of the decimal point. It does not change the value of numbers to the left of the decimal.

The difference between FIX and INT is that FIX does not return the next lower number when x is negative. (See also CINT.)

```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-28.87)
 28
Ok
```

FOR and NEXT Statements

FORMAT FOR **variable=x** TO **y** [STEP **z**]

.
.
.

NEXT **[variable][,variable]**...

PURPOSE Defines parameters for a loop.

■ ■ ■

variable is used as a counter. It must be an integer or in single precision.

x is the initial value of the counter.

y is the final value of the counter.

z is an integer or single-precision constant to be used as an increment.

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented with the step value, **z**. If STEP is not specified, the increment is assumed to be 1. A check is performed to see if the value of the counter is now greater than the final value **y**. If it is not greater, GWBASIC branches back to the statement after the FOR statement, and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop.

If **z** is negative, the test is reversed. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if **x** is already greater than **y** when **z** is positive, or if **x** is less than **y** when **z** is negative.

Nested Loops

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed inside another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for

the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. Using variable names on the NEXT statements causes programs to execute more slowly.

If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is displayed, and execution is terminated.

Example 1

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

The above shows a FOR...NEXT loop with a STEP value of 2.

Example 2

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3

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

The loop executes ten times. The final value for the loop variable is always set before the initial value is set.

FRE Function

FORMAT `v = FRE(x)`
`v = FRE(x$)`

PURPOSE Returns the number of bytes in memory not being used by GWBASIC, optionally tidying up the memory.

■ ■ ■

x and **x$** are dummy arguments and can be any value.

When all free memory has been used GWBASIC automatically performs a "garbage collection" to tidy up the data and free areas of memory that are no longer used. Since this may take some time, FRE(**x$**) can be used to force the collection process before returning the number of free bytes. Using FRE(**x$**) periodically will result in shorter delays for each garbage collection.

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

Your computer may give a different value depending on what options you have installed.

GET Statement (Files)

FORMAT GET [#]**file number** [,**record number**]

PURPOSE Reads a record from a random file into a random buffer.

■ ■ ■

file number is the number under which the file was opened.

record number must be in the range 1 to 16,777,215. If **record number** is omitted, the next record (after the last GET) is read into the buffer.

The LOF function can be used before a GET to see if that GET would go beyond the end-of-file marker. After a GET statement has been executed, you may use INPUT # and LINE INPUT # to read characters from the random file buffer. For additional information see Appendix A, Sequential and Random Files.

The GET and PUT statements allow fixed-length input and output for COM files. However, because of the low performance associated with telephone line communications, it is not advisable to use GET and PUT with COM files transmitted via telephone line.

```
50 CLS
60 INPUT "Name of the file to copy: ",IN.FILE$
70 INPUT "Name of the output file:  ",OUT.FILE$
80 OPEN "R",#1,IN.FILE$,128
90 FIELD #1, 128 AS IN.DATA$
100 'Check to see if the file exists - If it
     does then stop all action
110 OPEN "I",#2,OUT.FILE$
120 CLOSE #2
130 PRINT "File exists.  Cannot copy"
140 GOTO 330
150 'Print a message for the user
160 PRINT "Copying file"
170 OPEN "R",#2,OUT.FILE$,128
180 FIELD #2, 128 AS OUT.DATA$
190 'Read from file #1 and copy the information
     to file #2
200 GET #1
210 'Check to see if we are at end of file
220 'The error handler
```

```
230 IF EOF(1) = -1 THEN GOTO 280
240 LSET OUT.DATA$ = IN.DATA$
250 PUT #2
260 'and do it again!
270 GOTO 200
280 CLOSE
290 PRINT "Copy complete"
300 END
310 IF ERR = 53 AND ERL = 110 THEN CLOSE #2 :
    RESUME 160
320 PRINT : PRINT "Error";ERR;"at line";ERL
330 END
```

The above copies a file through the use of GET and PUT statements.

GET Statement (Graphics)

FORMAT GET **(x1,y1)-(x2,y2),arrayname**

PURPOSE Transfers a graphic image from the screen to an array.

■ ■ ■

(x1,y1)-(x2,y2) is a rectangle on the display screen. The rectangle is defined the same way as the rectangle drawn by the LINE statement using the ,B option: **(x1,y1)** is the upper left and **(x2,y2)** the lower right vertex.

arrayname is the name of the array that will hold the image. The array can be any type except string, and must be dimensioned large enough to hold the entire image (see below). Unless the array is type integer, the contents of the array after a GET will be meaningless when interpreted directly.

The GET statement transfers into the array the screen image bounded by the rectangle and described by the specified points.

The PUT statement transfers the image stored in the array onto the screen. Repeated PUTs to slightly different locations on the screen give the effect of animation. For more details on the use of GET and PUT for animation, see the PUT Statement (Graphics) later in this section.

To dimension an array, use the following formula to find the number of bytes required:

```
4+INT((x*bits+7)/8)*y
```

where:

x is the size of the x-dimension in pixels

y is the size of the y-dimension in pixels

bits is the number of bits used per pixel

The value of **bits** is 2 for medium resolution and 1 for high- and super resolution.

Assume you want to GET a 5 by 6 image into an integer array using medium resolution. The number of bytes required would be:

```
4+INT((5*2+7)/8)*6
```

or 16 bytes. Since the bytes per element of an array are:

2 for integer
4 for single precision
8 for double precision

you would need an integer array with at least 16/2, or 8 elements, a single precision array with at least 4 elements, or a double precision array with at least 2 elements.

It is possible to examine the x- and y-dimensions and even the data itself if an integer array is used. The x-dimension is in element 0 of the array, and the y-dimension is found in element 1. Remember that integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost) and then low byte.

GOSUB and RETURN Statements

FORMAT GOSUB **line**

.
.
.

RETURN **[line]**

PURPOSE Branches to and returns from a subroutine.

■ ■ ■

line in the GOSUB statement is the number of the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement causes GWBASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine.

line may be included in the RETURN statement to return to a specific line number from the subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the GOSUB will remain active, and errors such as "FOR without NEXT" may result.

Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

```
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
```

The statements in the above example are executed in the following sequence:

10 Calls a subroutine
40 Prints a string
50 Prints a string
60 Prints a string
70 Ends the subroutine
20 Prints a string
30 Ends the program

GOTO Statement

FORMAT GOTO **line**

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

■ ■ ■

line is the number of a line in the program.

If **line** is the number of an executable statement, that statement and those following are executed. If **line** refers to a nonexecutable statement (such as REM or DATA), the program continues at the first executable statement encountered after **line**.

GOTO can be useful in debugging to re-enter a program at a desired point in direct mode.

Use ON...GOTO to branch to different lines based on the result of an expression.

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

Line 50 passes control to line 5, which is a nonexecutable statement. The READ is then invoked, and the program continues in a loop until the data is exhausted and a read error is flagged in line 10.

GWBASIC Command

FORMAT GWBASIC [<stdin] [>[>stdout]] [filespec] [/C:combuffer] [/D] [/F:files] [/I] [/M:[max workspace][,max block size]] [/S:bsize]

NOTE: If you type BASIC or BASICA,the program looks for and loads GWBASIC.EXE. If GWBASIC.EXE is not found, an error message is displayed.

PURPOSE Causes GWBASIC to run under the specified conditions from the DOS command level.

■ ■ ■

stdin and **stdout** are used for redirecting input and output - see Section 3, under "Redirection of Input and Output". They specify files for a BASIC program to respectively read input from and write output to. If used, these parameters must be given before any of the switches (/C, /D, etc.)

filespec identifies a program to be loaded and executed, and has the form shown under "File and Device Information" in Section 3. If no extension is supplied and the length of the file name is eight characters or less, .BAS is used as the default extension. This option causes the same action as RUN "**filespec**". You can use this format of the GWBASIC statement to run BASIC programs in batch mode by putting these GWBASIC statements in an AUTOEXEC.BAT file. Programs run in this way must exit via the SYSTEM command each time.

/C:**combuffer** is for use with the asynchronous communications adapter and sets the size of the communications buffer that is used for receiving data. (The buffer for transmitting data is always set for 128 bytes.) The maximum value that may be entered for this option is 32767. If the option is omitted, the receive buffer is set for 256 bytes. For a high-speed line the suggested value is /C:1024. If you have more than one asynchronous communications adapter, this option sets the size for all receive buffers. Entering /C:0 disables RS232 support.

/D causes the double-precision math package to stay resident in memory. If you omit /D, double-precision functions are ignored and the memory space (about 3K) is freed for program use.

/F:**files** sets the number of files that may be open at any one time during the execution of a GWBASIC program, and is only meaningful if the /I switch is also set. Each file requires 62 bytes of memory for its control block, plus 128 bytes for the data buffer, unless this value is changed with the /S: option. If the /F: option is omitted, **files** is set to 3. The maximum recommended value is 10.

/I causes GWBASIC to statically allocate space for file operations based on the /S and /F switches, which have no effect unless /I is specified.

/M is used as follows. **max workspace** sets the maximum number of bytes that may be used as GWBASIC work space. Since GWBASIC uses only a maximum of 64 Kbytes of memory, that is the highest value that may be set (hex FFFF). This option can be used to reserve space for assembly language subroutines or for special data storage. If this option is omitted, all available memory is used (up to 64K). **max block size** specifies the maximum number of 16-byte blocks of memory to be reserved for BASIC and any assembly language routines you want to run. This is necessary when using the SHELL statement, otherwise COMMAND.COM will be loaded on top of the assembly language routines when SHELL is executed. Thus to reserve 64 Kbytes for BASIC and 512 bytes for assembly language routines, specify: /M:,4128 giving 4096 blocks for BASIC and 32 blocks for the routines.

files, bsize, combuffer, and **max workspace** may be decimal, octal (preceded by &O), or hexadecimal (preceded by &H) numbers.

/S:**bsize** sets buffer size for use with random files, and is only meaningful if the /I switch is also set. The parameter for record length on the OPEN statement may not exceed this value. Default buffer size is 128 bytes. The maximum value that may be entered is 32767. Using /S:512 is suggested for improved performance with random files since this matches the physical sector size on the diskette.

Below are some examples of the GWBASIC command:

```
GWBASIC PAYROLL.ABC
```

GWBASIC will use all memory, up to three files at any one time, and the program PAYROLL.ABC will be loaded and executed.

```
GWBASIC INVENT/F:6
```

GWBASIC will use all memory, up to 6 files, and the program INVENT.BAS will be loaded and executed.

GWBASIC /M:32768

The maximum size of the work space is set to be 32768, or 32 Kbytes of memory, and no more than three files will be used at a time. No program is run, and the "Ok" prompt will appear.

GWBASIC /M:32000,2048

Allocates 32 Kbytes maximum (a total of 32,768 bytes) but BASIC will only use the lower 32,000, allowing 768 bytes for assembly language routines.

GWBASIC TTY/C:512

GWBASIC uses all of memory and three files. The RS232 receive buffer is allocated 512 bytes and the transmit buffer 128 bytes. The program TTY.BAS is loaded and executed.

HEX$ Function

FORMAT `v = HEX$(x)`

PURPOSE Returns a string that represents the hexadecimal value of the decimal argument.

■ ■ ■

`x` is an integer.

See the OCT$ function for information on octal conversion.

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

IF Statement

FORMAT IF **expression** [,] THEN **clause** [ELSE **clause**] [:**statement...**]

IF **expression** [,] GOTO **line** [[,] ELSE **clause**]

PURPOSE Performs a branch or executes one or more statements if a specified condition is satisfied.

■ ■ ■

expression is a numeric expression.

clause is a GWBASIC statement, a sequence of statements (separated by colons), or the number of a line to branch to.

line is the number of a line existing in the program.

Execution of the first of the two forms above is performed according to the table below.

Table 7-5

EXECUTION OF IF-THEN-ELSE STATEMENTS

Expression		ELSE Clause Present	Clause to be Executed	Continue Execution on Same Line
Logical	Value			
TRUE	<>0	YES	THEN	NO
TRUE	<>0	NO	THEN	YES
FALSE	=0	YES	ELSE	YES
FALSE	=0	NO	Next Statement	NO

If the result of **expression** is true (not zero), the THEN or GOTO clause is executed. THEN is followed by a clause that can be either a line number to branch to, or one or more statements to be executed. GOTO is always followed by a line number.

If the result of **expression** is false (zero), the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed (the ELSE clause follows the same rules as the THEN clause).

Execution then continues with the next executable statement after IF.

NOTE: When using IF to test equality for a value that is the result of a single or double-precision computation, remember that the internal representation of the value may not be exact. This is because single- and double-precision values are stored internally in floating point binary format. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use

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

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

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example:

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

is a valid statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

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

will not print "A<>C" when A<>B.

If you enter an IF...THEN statement in direct mode and it directs control to a line number, then an "Undefined line number" error results unless you already entered the line with the specified number in indirect mode.

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

This statement gets record I if I is not zero.

```
100 IF (I>10) and (I<20) THEN DB=1985-1: GOTO
300 ELSE PRINT "OUT OF RANGE"
```

If I is between 10 and 20, DB is calculated and execution branches to line 300. If I is not in this range, the message "OUT OF RANGE" is printed. Note the use of two statements in the THEN clause.

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

This statement causes printed output to go to either the screen or printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the printer; otherwise, output goes to the screen.

INKEY$ Function

FORMAT v$ = INKEY$

PURPOSE Reads a character from the standard input device (normally the keyboard unless input is redirected).

■ ■ ■

INKEY$ returns either the actual character received from the keyboard, or a null string if no character has been typed. In some cases the character is returned as a two-character string that indicates an extended character code. Information on the characters that return these codes is given in Table C-2.

The value returned by INKEY$ must be assigned to a string variable before it can be processed by GWBASIC.

```
50  CLS
60  LOCATE 10,10
70  PRINT "Press RETURN to continue ";
80  LOCATE 10,30
90  RET$ = INKEY$
100 IF RET$ = "" THEN GOTO 80
110 IF ASC(RET$) <> 13 THEN GOTO 80
120 END
```

The above uses the INKEY$ function to poll the keyboard until the RETURN key, which has an ASCII value of 13, is pressed.

INP Function

FORMAT **v** = INP(**n**)

PURPOSE Returns the byte read from port address **n**.

■ ■ ■

n is an integer in the range 0 to 1023 (&H3FF).

For examples of using the INP function with a communications program, see Section 6 under "Accessing the Registers".

A port address map is given in Table 7-6.

INP is the complementary function to the OUT statement (see OUT statement).

```
100 A=INP(&H3DA)
```

A byte is read from port address &H3DA and assigned to variable A.

Table 7-6

PORT ADDRESS MAP

Hex Range	Address Bit										Device
	9	8	7	6	5	4	3	2	1	0	
00-0F	0	0	0	0	0	*	A3	A2	A1	A0	DMA Chip 8237-2
20-21	0	0	0	0	1	*	*	*	*	A0	Interrupt 8259A
40-43	0	0	0	1	0	*	*	*	A1	A0	Timer 8253-5
60-63	0	0	0	1	1	*	*	*	A1	A0	PPI 8255-5
80-83	0	0	1	0	0	*	*	*	A1	A0	DMA Page Registers
AX**	0	0	1	0	1						NMI Mask Registers
CX	0	0	1	1	0						Reserved
EX	0	0	1	1	1						Reserved
200-20F	1	0	0	0	0	0	A3	A2	A1	A0	Game I/O Adapter
278-27F	1	0	0	1	1	1	1	*	A1	A0	Reserved
2F8-2FF	1	0	1	1	1	1	1	A2	A1	A0	Reserved
300-307	1	1	0	0	0	0	0	A2	A1	A0	Hard Disk I/O Ports
378-37F	1	1	0	1	1	1	1	*	A1	A0	Auxiliary Parallel Printer Port
3B0-3BF	1	1	1	0	1	1	A3	A2	A1	A0	Built-in Parallel Printer Port and Mono-chrome Display
3D0-3DF	1	1	1	1	0	1	A3	A2	A1	A0	Color/ Graphics Adapter

Table 7-6 (Cont.)

Hex Range	Address Bit										Device
	9	8	7	6	5	4	3	2	1	0	
3F0-3F7	1	1	1	1	1	1	0	A2	A1	A0	5 1/4" Drive
3F8-3FF	1	1	1	1	1	1	1	A2	A1	A0	RS-232C (Serial Port)

*Not in decode.

**At power-on, the nonmaskable interrupt, NMI, to the 8088 is disabled by external hardware. This can be set and reset as follows:

To set mask: Write 80H to I/O Address 0A0H

To clear mask: Write 00H to I/O Address 0A0H

INPUT Statement

FORMAT INPUT[;]["prompt";]variable[,variable]...

PURPOSE Receives input from the standard input device (normally the keyboard unless input has been redirected) during program execution.

■ ■ ■

"**prompt**" is a string constant that can be used to prompt for the desired input.

variable is the name of the numeric or string variable or array element that will receive the input.

When an INPUT statement occurs, the program will pause and display a question mark on the screen to indicate it is waiting for data. If **prompt** is included, it is displayed before the question mark. You may then type in the required data. The string that you input must not contain delimiters such as a comma (,). If you want to include delimiters in the input string, use the LINE INPUT statement.

Use a comma instead of a semicolon after the prompt string to suppress the question mark. For example, the statement:

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

will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, the cursor will remain on the same line as your response, and subsequent output from the program will begin at this point. If you omit the semicolon, subsequent output will begin on the next line, as in the example below.

The data you enter is assigned to the variable(s) given in the variable list. The number of data items must be the same as the number of variables in the list. Data items must be separated by commas.

The type of each data item entered must agree with the type specified by the variable name, otherwise the message "Type mismatch" will be displayed. (Strings entered in response to an INPUT statement need not be surrounded by quotation marks.)

If you respond to INPUT with too many or too few items or with the wrong type of value (letters instead of numbers, etc.), GWBASIC displays the message "?Redo from start", and does not assign any of the input values to variables until you give an acceptable response.

Press Ctrl-Break or Ctrl-C to stop INPUT. GWBASIC returns to command level, displaying the "Ok" prompt. Typing CONT resumes program execution at the INPUT statement.

```
10 INPUT A$
20 PRINT "STRING = "A$
30 END
RUN
? HELLO
 STRING = HELLO
Ok
```

The question mark in the above example shows that the program is waiting for input to be typed on the keyboard.

INPUT$ Function

FORMAT `v = INPUT$(n[,[#]filenum])`

PURPOSE Reads a specified number of characters from the standard input device (normally the keyboard unless input has been redirected) or from file number **filenum**.

■ ■ ■

n is the number of characters to be read from the file.

filenum is the file number used on the OPEN statement. The # sign is optional. If **filenum** is omitted, input is read from the standard input device.

If the keyboard is used for input, no characters will be displayed on the screen. All characters (including control characters) are passed through except Ctrl-Break, which is used to interrupt the execution of the INPUT$ function. When using the keyboard for input in response to INPUT$, there is no need to press <Return>.

When working with communication files, INPUT$ is preferred over other input statements because it will pass all characters except Ctrl-Break. LINE INPUT terminates when a carriage return is read, and INPUT terminates on encountering a comma or carriage return.

The following program uses INPUT$ to print a hexadecimal dump of a file.

```
40 CLS
50 INPUT "File to dump: ",F$
60 HX$ = SPACE$(2)
70 OPEN "1",#1,F$
80 IF EOF(1) THEN GOTO 130
90 CHAR$ = INPUT$(1,#1)
100 LSET HX$ = RIGHT$("0"+HEX$(ASC(CHAR$)),2)
110 PRINT HX$;" ";
120 GOTO 80
130 CLOSE
140 END
```

The following routine uses INPUT$ to prompt for a response from the keyboard:

```
210 PRINT "Continue (Y/N)?"
220 A$ = INPUT$(1)
230 IF A$ = "Y" THEN 250
240 IF A$ = "N" THEN 330
      .
      .
      .
```

INPUT# Statement

FORMAT INPUT#**filenum,variable[,variable]...**

PURPOSE Reads data items from a sequential device or file (as well as a random file) and assigns them to program variables.

■ ■ ■

filenum is the number used when the file was OPENed for input.

variable is the name of a variable that will have an item in the file assigned to it. It may be a string or numeric variable, or an array element.

The type of data in the file must match that specified by the variable name. Unlike INPUT, no question mark is displayed with INPUT#.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a number. The number ends with a space, carriage return, line feed, or comma.

To read a string that contains delimiters (such as ",") from the file, use the LINE INPUT # statement instead.

If GWBASIC is scanning the data for a string item, it will also ignore leading spaces, carriage returns, and line feeds. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not 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. It will end with a comma, carriage return, or line feed, or after 255 characters have been read. If end-of-file is reached while a numeric or string item is being input, the item is cancelled.

For an example, see Appendix A.

INSTR Function

FORMAT **v = INSTR([n,]a$,b$)**

PURPOSE Searches for the first occurrence of string **b$** within string **a$**, and returns the position at which the match is found. The start position within **a$** for the search can optionally be specified by **n**.

■ ■ ■

n must be in the range 1 to 255, otherwise an "Illegal function call" message will be displayed.

a$,b$ may be string variables, string expressions, or string constants.

If **n** is greater than the number of characters in **a$** (i.e., if **n** > LEN(**a$**)), or if **a$** is null, or if **b$** cannot be found, INSTR returns 0. If **b$** is null, INSTR returns **n** (or 1 if **n** is not specified).

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

This program looks for B$ within A$, or in this example, for "B" within "ABCDEB". "B" will be found in position 2 if the search starts at the beginning of A$. If the search starts in the fourth position, "B" is discovered in the sixth position.

INT Function

FORMAT `v = INT(x)`

PURPOSE Returns the largest integer less than or equal to **x**.

■ ■ ■

x is any numeric expression.

See the FIX and CINT functions, which also return integer values.

```
PRINT INT(99.89)
 99
Ok

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

The above examples show how INT always returns a whole number smaller than the decimal number.

IOCTL Statement

FORMAT IOCTL **[#]filenum, string**

PURPOSE Transmits a command string to a device driver.

■ ■ ■

[#]filenum is the file number of the device driver, as specified when the driver was OPENed. Note that the driver must be open before IOCTL is used, otherwise the statement will have no effect.

string is a string expression containing control characters or commands for the device driver. The string can be up to 255 characters long, and commands within the string must be separated by semicolons.

Most standard DOS device drivers cannot process IOCTL strings, so the statement is used mainly for user-installed device drivers (see under that heading in Section 3). You have to ensure that the installed driver is written so that it will recognize any IOCTL commands sent to it. Typical commands consist of 2 or 3 characters, optionally followed by an alphanumeric argument, e.g. PL66 to set the page length to 66 lines.

If you had a user-installed device driver replacing the standard DOS printer driver LPT1 and you wanted to set the page length to 66 lines per page on LPT1, then assuming that the new driver was able to set page lengths, the procedure would be:

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

See also the IOCTL$ Function.

IOCTL$ Function

FORMAT `v$ = IOCTL$ ([#]filenum)`

PURPOSE Returns a command string from a device driver.

■ ■ ■

`[#]filenum` is the file number of the device driver, as specified when the driver was OPENed. Note that the driver must be open, otherwise IOCTL$ will have no effect.

The IOCTL$ function is most frequently used to receive acknowledgement that an IOCTL statement succeeded or failed.

IOCTL$ could also be used to obtain device status information, such as asking a communications device to return the current baud rate, information on the last error, logical line width, etc.

As an example, you could use the IOCTL statement to set the page length to 66 lines for printer LPT1: and retrieve the length so that IOCTL$ can test for it (remember that the device driver must be able to recognize the commands in the IOCTL string):

```
10 OPEN "\DEV\LPT1" AS #1
20 IOCTL #1, "PL66;GP"
```

Continuing this example, you can use the IOCTL$ function to check whether the page length command was successful, and close the printer driver file if the command failed:

```
30 IF IOCTL$(1) <> "66" THEN CLOSE 1
```

See also the IOCTL statement.

KEY Statement

FORMAT KEY **n,x$**

KEY **n**, CHR$(**mask**) + CHR$(**scan**)

KEY LIST

KEY ON

KEY OFF

PURPOSE Sets or displays the values of the function keys, or sets values for a user-defined key or key sequence.

■ ■ ■

n is a numeric expression in the range either 1 to 10 to identify a function key, or 15 to 20 to identify a user-defined key or key sequence.

x$ is a variable containing text to be assigned to a function key, or a string containing this text, which may be up to 15 characters long (if the text is longer than this, only the first 15 characters will be displayed). Note that string constants must be enclosed in quotes. Assigning a null string to a function key disables that key.

mask is a numeric expression and indicates that one or more of the following keys is to form part of a user-defined key sequence: Caps Lock, Num Lock, Alt, Ctrl, and the two Shift keys. The possible values for **mask** in hexadecimal are:

&H40 - Caps Lock
&H20 - Num Lock
&H08 - Alt
&H04 - Ctrl
&H02 - left-hand Shift key
&H01 - right-hand Shift key

scan is a numeric expression identifying a key on the keyboard that is to form the remainder of the user-defined key sequence. **scan** is specified in the form of a "scan code"; the code for each key is given in Table F-1.

The initial values of the function keys are given in Table 7-7.

Table 7-7

FUNCTION KEY VALUES

Key	Value	Key	Value
F1	LIST	F2	RUN + CHR$(13)
F3	LOAD"	F4	SAVE"
F5	CONT + CHR$(13)	F6	,"LPT"
F7	TRON + CHR$(13)	F8	TROFF + CHR$(13)
F9	KEY	F10	SCREEN 0,0,0

KEY **n**,**x$** causes the text referenced by **x$** to be assigned to function key **n**. The text is input to GWBASIC whenever that key is pressed.

KEY **n**, CHR$(**mask**) + CHR(**scan**) causes the key sequence identified by **mask** and **scan** to be assigned as user-defined key **n**. See below under "Trapping User-Defined Keys".

KEY LIST lists the values of each key on screen. All 15 characters of each value are given.

KEY ON causes the function key values to be displayed on the 25th line. When the width is 40, five of the ten keys are displayed, and when the width is 80, all ten are displayed. However, only the first six characters of each value are displayed. ON is the default state for the function key display.

Note that entering a width of 40 causes five keys to be displayed instead of ten during medium-resolution graphics emulation (see SCREEN statement for information on emulation modes). It has the same effect on displays with color/graphics monitor adapters in both 40-column text mode and medium-resolution graphics mode.

Also note that you must press Ctrl-T to display the remainder of the keys when the width is 40.

KEY OFF erases the key display from the 25th line. The function keys are not disabled.

When a function key is assigned, INKEY$ will return one character of the function key each time it is called. If the function key is disabled, INKEY$ returns a two-character string, the first of which is binary zero and the second the key code. Appendix C, ASCII Character Codes, lists this code.

Information on line 25 of the screen is not scrolled. If you have used KEY OFF, you may display information on line 25 by using LOCATE 25,1 followed by PRINT.

Trapping User-Defined Keys

Use the format KEY **n**, CHR$(**mask**) + CHR$(**scan**) to identify a key sequence that can be trapped by means of subsequent ON KEY (**n**)... and KEY (**n**) ON statements. Once a key sequence is trapped, the program can branch to a subroutine and perform any desired processing (see the last example below).

You can also trap "super-shifted" keys, e.g. Ctrl-Shift-D or Ctrl-Alt-D. The statement:

```
KEY n,CHR$(&H04 + &H08) + CHR$(&H20)
```

will trap the sequence Ctrl(&H04)-Alt(&H08)-D(&H20). You could use this form, for example, to trap a Ctrl-Break or Ctrl-Alt-Del sequence to prevent an inexperienced user from accidentally interrupting a program, or rebooting the system while the program was running. Note, however, that the trap routine would have to include a test for this key sequence.

The following applies when keys are trapped:

1. Ctrl-PrtSc is processed first. Characters will still be sent to the printer even if Ctrl-PrtSc is trapped.

2. Function keys and cursor direction keys are processed next. Designating these keys as user-defined keys has no effect, however, since they are considered to be predefined.

3. The user-defined key(s) are then processed.

4. If a key is trapped, it is not passed on as character input to the BASIC program itself.

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

Assigns the string "MENU" + Enter to function key 1.

```
40 KEY 1,""
```

Disables function key 1.

```
10 KEY OFF 'Turns off display during initialization
20 DATA "EDIT", "LET", "SYSTEM", "PRINT", "LPRINT"
30 FOR A = 1 TO 5
40   READ FUNKEY$(A)
50   KEY A, FUNKEY$(A)
60 NEXT A
70 KEY ON 'Displays new key values
```

Assigns new values to the first five function keys.

```
10 KEY 20, CHR$(&H04) + CHR$(&H20)
20 ON KEY(20) GOSUB 100
30 KEY(20) ON
   .
   .
   .
90 END
100 PRINT "You pressed Ctrl-D":RETURN
```

Traps the key sequence Ctrl-D and performs the processing at line 100 when that sequence is recognized.

KEY(n) Statement

FORMAT KEY(**n**) ON

KEY(**n**) OFF

KEY(**n**) STOP

PURPOSE Enables and disables trapping a specified key.

■ ■ ■

n is the number of the key to be trapped:

1-10	Function keys F1-F10
11	Cursor up
12	Cursor left
13	Cursor right
14	Cursor down
15-20	User-defined keys

KEY(**n**) ON activates trapping of the specified key. The action to be taken by the program when the specified key is pressed is determined by a subsequent ON KEY(**n**)... statement, which branches to a subroutine.

KEY(**n**) STOP stops trapping, but if the key is subsequently pressed, this statement will cause an immediate trap if trapping is turned back on.

KEY(**n**) OFF not only stops trapping, but if a key is subsequently pressed, does not remember it.

This statement has no effect with the INPUT or INKEY$ statements. A different trap routine for each key must be created for use with these statements.

NOTE: KEY(**n**) ON and KEY ON are two different statements with very different meanings. See the KEY statement for details of KEY ON.

```
40 CLS
50 'Enable function keys 1 to 4
60 FOR A = 1 TO 4
70 KEY(A) ON
80 NEXT A
90 'Disable function keys 5 to 10 and the
    arrow keys (up, left, right, down)
100 FOR A = 5 TO 14
110 KEY(A) OFF
120 NEXT A
130 'Determine which subroutine to use if
     a function key is pressed
140 ON KEY(1) GOSUB 240
150 ON KEY(2) GOSUB 250
160 ON KEY(3) GOSUB 260
170 ON KEY(4) GOSUB 270
180 'Print a message
190 LOCATE 10,10
200 PRINT "Press F1, F2, F3, or F4";
210 LOCATE 20,20 : PRINT " ";
220 GOTO 210
230 'Subroutines
240 PRINT "You pressed F1"; : RETURN
250 PRINT "You pressed F2"; : RETURN
260 PRINT "You pressed F3"; : RETURN
270 PRINT "You pressed F4"; : RETURN
```

This program demonstrates use of the KEY ON, KEY OFF, and ON KEY(n) statements.

KILL Command

FORMAT KILL **filespec**

PURPOSE Deletes a file from a disk.

■ ■ ■

filespec is a file specification (see Section 3, under "File and Device Information") in the form of a string constant, and consists of a filename or pathname with an optional device name. If used, the device name must be that of a diskette or hard drive. If the device name is omitted, the DOS default drive is used.

KILL is similar to the ERASE command in DOS. It requires using the extension, if one exists.

If a file is open when a KILL command is executed, a "File already open" error will result. The same error will occur if you try to delete a file that has the same name as an open file in another directory, even though they are two completely different files.

```
10 KILL "A:TEST.BAS"
```

This example will delete the file TEST.BAS from drive A:.

```
10 KILL "..\SALES\MARY\PROG1.EXE"
```

The above accesses the subdirectory MARY in the directory SALES and deletes the file PROG1.EXE.

LCOPY Statement

FORMAT LCOPY **n**

PURPOSE Prints the current screen contents.

■ ■ ■

n is an integer expression from 0 to 2 as follows:

0 - both text and graphics are printed
1 - graphics only are printed
2 - text only is printed

Using LCOPY within a program has the same effect as pressing Shift-PrtSc, causing the contents of the screen at that point to be sent to the printer.

LEFT$ Function

FORMAT `v$ = LEFT$(x$,n)`

PURPOSE Returns a string comprising the leftmost **n** characters of **x$**.

■ ■ ■

n must be in the range 0 to 255. If **n** is greater than the number of characters in **x$** (i.e., if **n** > LEN(**x$**)), the entire string (**x$**) will be returned. If **n** = 0, a null string (length zero) is returned.

See also the MID$ and RIGHT$ functions.

```
10 A$="BASIC"
20 B$=LEFT$(A$,3)
30 PRINT B$
RUN
 BAS
```

The first three characters of "BASIC" are printed.

LEN Function

FORMAT **v = LEN(x$)**

PURPOSE Returns the number of characters in **x$**.

■ ■ ■

Nonprinting characters and blanks are included in the number of characters counted.

```
10 X$ = "WESTLAKE VILLAGE, CA"
20 PRINT LEN(X$)
RUN
 20
Ok
```

There are 20 characters in the string "WESTLAKE VILLAGE, CA" because the comma and the blanks are counted.

LET Statement

FORMAT [LET] **variable=expression**

PURPOSE Assigns the value of an expression to a variable.

■ ■ ■

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

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

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

The two programs above are equivalent.

LINE Statement

FORMAT LINE [[STEP](**x1,y1**)] - [STEP](**x2,y2**) [,[**color**] [,B[F]]][,**style**]

PURPOSE Draws a line or a box on the screen.

■ ■ ■

This statement can only be used in graphics mode (SCREEN 1, 2, 104 or 105).

(**x1,y1**) is the coordinate for the starting point of the line.

(**x2,y2**) is the ending point for the line.

color is the number of the color in which the line is to be drawn. If the ,B or ,BF option is used, the box is drawn in this color. In medium resolution (SCREEN 1), **color** ranges from 0 to 3, where 0 indicates the background color and 1 to 3 denote colors from the current palette (see Table 7-2). In high resolution (SCREEN 2) and super resolution (SCREEN 104 and 105), **color** can be either 0 (background color) or 1 (foreground color). Default is 3 for medium resolution and 1 for high- and super resolution. Default for monochrome screens is the foreground color.

B draws a box in the foreground, with the points (**x1,y1**) and (**x2,y2**) as opposite corners.

BF draws a filled box in the foreground.

style is an integer that specifies a pattern to be used for the line, allowing you to draw lines made up of dots, dashes or a combination of both. The pattern appears on the screen as a 16-bit binary representation of the integer, using the binary 1s to set a pixel on. Values should be given in hexadecimal; thus for example a value of &HFF00 for **style** would cause a dashed line to be drawn, since the binary representation of &HFF00 (decimal 65280) in 16 bits is 11111111 00000000. Note that **style** has no effect on filled boxes.

When out-of-range coordinates are given, the coordinate that is out of range is given the closest legal value.

The coordinates form STEP (**xoffset,yoffset**) can be used in place of an absolute coordinate. For example, assume that the most recent point

referenced was (0,0). The statement LINE STEP (10,5) would specify a point at offset 10 from x and offset 5 from y.

If the STEP option is used for the second coordinate on a LINE statement, it is relative to the first coordinate in the statement. Other ways to establish a new "most recent point" are to initialize the screen with the CLS and SCREEN statements, or to use PSET, PRESET, CIRCLE or DRAW.

```
5 SCREEN 105
10 LINE -(200,200)
```

Draws a line from the last point referenced to 200,200 in the foreground color. This is the simplest form of the LINE statement.

```
20 LINE (0,0)-(639,300)
```

Draws a diagonal line across the screen (downward).

```
30 LINE (0,100)-(639,100)
```

Draws a line horizontally across the screen.

```
40 LINE (10,10)-(200,200),2
```

Draws a line in color 2.

```
10 ON ERROR GOTO 50
20 CLS
30 LINE -(RND*639,RND*300),RND*4
40 GO TO 20
50 IF ERR=5 THEN RESUME 30
```

Draws lines forever using random attributes. The following statements:

```
10 FOR X=0 to 639
20 LINE (X,0)-(X,300),X AND 1
30 NEXT
```

draw an alternating line on - line off pattern.

```
10 LINE (0,0)-(100,100),,B
```

Draws a box in the foreground. (Note that color is not included.)

```
20 LINE STEP (0,0)-STEP (200,200),2,BF
```

Draws a filled box in the foreground in color 2. Coordinates are given as offsets.

```
10 LINE (0,0) - (319,160),,,&HAAAA
```

Draws a dotted line from the top left towards the center of the screen.

LINE INPUT Statement

FORMAT LINE INPUT[;]["**prompt**";]**stringvar**

PURPOSE Inputs an entire line (up to 254 characters), ignoring delimiters, to a string variable.

■ ■ ■

LINE INPUT and LINE INPUT# allow you to input strings containing delimiters (such as "," or ".") to a program; the INPUT and INPUT# statements do not permit delimiters in the input string.

"prompt" is a string constant displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string.

stringvar is the name of the string variable or array element to which the line will be assigned. All input from the end of **"prompt"** to the carriage return is assigned to **stringvar**. However, if a linefeed/carriage return sequence (this order only) is encountered, both characters are echoed, but the carriage return is ignored, the linefeed is put into **stringvar**, and input continues.

If LINE INPUT is immediately followed by a semicolon, then if you press <Return> to terminate the input line when the program is running, the cursor will not move to the start of the next line but will stay where it is.

Press Ctrl-Break or Ctrl-C to stop LINE INPUT. GWBASIC returns to command level, displaying the "Ok" prompt. Typing CONT resumes execution at the LINE INPUT statement.

When GWBASIC is invoked with redirected input, all LINE INPUT statements will read data from the specified input file instead of from the keyboard (see "Redirection of Input and Output" in Section 3).

When input is redirected, GWBASIC continues to read from the new source until a Ctrl-Z is encountered (this condition can be tested by the EOF function). If the file is not terminated by a Ctrl-Z, or if a program tries to read past the end-of-file, then any open files are closed, the message "Read past end" is output, and the system exits to DOS.

For an example, see the LINE INPUT# statement.

LINE INPUT # Statement

FORMAT LINE INPUT #**filenum,stringvar**

PURPOSE Reads an entire line (up to 254 characters) from a sequential file, ignoring delimiters, to a string variable. It may also be used for random files.

■ ■ ■

filenum is the number under which the file was OPENed.

stringvar is the name of a string variable to which the line will be assigned.

LINE INPUT# reads all characters in the sequential file until it reaches a carriage return/line feed sequence, which it omits. Note that, if encountered, a line feed/carriage return sequence is returned as part of the string.

LINE INPUT# is especially useful if each line of a file has been broken into fields, or if a GWBASIC program saved in ASCII mode is being read as data by another program. (See SAVE command.)

See Appendix A, Sequential and Random Files.

```
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
```

In this example, the program writes the customer information to a sequential file. After closing and reopening the file, it reads the information back using the LINE INPUT # statement.

LIST Command

FORMAT 1 LIST **[line1]**

FORMAT 2 LIST **[line1][-[line2]][,"device"]**

PURPOSE Lists all or part of the program currently in memory.

■ ■ ■

line is in the range 0 to 65529.

device is a string expression such as SCRN: or LPT1: and indicates the device to which the listing is to be sent. Note that the string expression must be enclosed in quotes.

Format 1

If **line** is omitted, the program is listed beginning at the lowest line number. If **line** is included, only the specified line will be listed.

Format 2

This format allows the following options:

1. If only the first **line** is specified and followed by a hyphen, that line and all higher-numbered lines are listed.
2. If only the second **line** is specified with a hyphen preceding **line**, all lines from the beginning of the program through that line are listed.
3. If both **lines** are specified, the entire range is listed.
4. If **device** is omitted, the listing is shown on the screen.

Use Ctrl-Break to interrupt LIST.

GWBASIC always returns to command level after LIST.

LIST

Lists the program currently in memory.

```
LIST 500
```

Lists line 500.

```
LIST 150-
```

Lists all lines from 150 to the end.

```
LIST -1000
```

Lists all lines from the lowest number through 1000.

```
LIST 150-1000
```

Lists lines 150 through 1000, inclusive.

```
LIST 150-1000,"LPT1:"
```

Lists lines 150 through 1000 on the line printer.

LLIST Command

FORMAT LLIST `[line1][-[line2]]`

PURPOSE Lists all or part of the program currently in memory on the printer (LPT1:).

■ ■ ■

LLIST assumes a 132-character-wide printer.

LLIST follows the options of LIST, Format 2.

GWBASIC returns to command level after LLIST.

For an example, see the LIST command.

LOAD Command

FORMAT LOAD **filespec[,R]**

PURPOSE Loads a program from a specified device into memory and, optionally, runs it.

■ ■ ■

filespec is a file specification (see Section 3, under "File and Device Information") in the form of a string constant, and identifies the program to be loaded. It consists of a filename or pathname with an optional device name. If used, the device name must be that of a diskette or hard drive. If the device name is omitted, the DOS default drive is used.

If no extension is supplied and the filename is eight characters or less, the extension .BAS is added to the filename.

The R option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program.

If the R option is used, the program is run after it is loaded, and all open data files are kept open. Thus, LOAD with the R option may be used to chain several programs (or segments of the same program), and information may be passed between the programs using data files.

```
LOAD "B:MYPROG"
```

The program named "MYPROG" located on drive B: is loaded into memory.

```
LOAD "STRTRK",R
```

The program named "STRTRK" is loaded and run.

LOC Function

FORMAT **v = LOC(filenum)**

PURPOSE Returns the current position in the file.

■ ■ ■

filenum is the number used when the file was opened.

With random files, LOC returns the record number just read or written.

With sequential files, LOC returns the number of records (128-byte blocks) read from or written to the file since it was opened. This number will be 1 when the file is opened since GWBASIC reads the first sector of the file.

If the file was opened in append (A) or output (O) mode, LOC returns the size of the file in (bytes/128).

With a communications file, LOC displays the number of characters in the input buffer waiting to be read. If there are more than 255 characters, LOC returns 255. (The default size is 256 characters, but this can be changed with the /C: option of the GWBASIC command.) This eliminates the need for testing for string size before reading data into the buffer since a string is limited to 255 characters.

If there are fewer than 255 characters in the input buffer waiting to be read, LOC returns the number of characters that can be read from the communications device. Note that if the device was opened in ASCII mode (see the OPEN "COM..." statement), character queueing stops as soon as end-of-file is received. The end-of-file marker itself is not queued and cannot be read. Any attempt to read past the end-of-file will result in an "Input past end" error.

```
200 IF LOC(1)>50 THEN STOP
```

The program is stopped after the 50th record in the file.

LOCATE Statement

FORMAT LOCATE **[y][,[x][,[cursor][,[start][,stop]]]]**

PURPOSE Moves the cursor to a specified position on the screen. Optional parameters turn the blinking cursor on and off and define its size.

■ ■ ■

y is a numeric expression with a value in the range 1 to 25 indicating the number of the screen line where the cursor should be placed.

x is a numeric expression with a value ranging either from 1 to 40 or 1 to 80 (depending on the number of display columns) to indicate the number of the screen column where the cursor should be placed.

cursor is 0 to indicate the cursor is off (invisible) or 1 to indicate it is on (visible). Note that the cursor is initially off when a program runs.

start is a numeric expression to indicate the cursor starting scan line (for more information on **start** and **stop**, see Section 4).

stop is a numeric expression to indicate the cursor stop scan line.

If you have both text and graphic pages, remember that LOCATE moves the cursor in the active page of the mode currently being used.

The last three parameters (**cursor**, **start**, and **stop**) may only be used if the system is in text mode (see the SCREEN statement).

If values are entered outside the ranges given, an "Illegal function call" error results, and previous values are retained.

start and **stop** enable you to make the cursor any size within its character position by indicating the beginning and ending scan lines. These lines are numbered from 0 at the top of the character position to 12 or 15 at the bottom, depending on your system (see Section 4). Note that the bottom position is 7 on most compatible color/graphics monitor adapters. No range checking is performed for the value of **stop**.

If **start** is given without **stop**, **stop** assumes the value of **start**. If **start** is greater than **stop**, there will be a two-part cursor, in which

the cursor wraps from the bottom line back to the top. If the two values are equal, the cursor will be a thin line at whatever position is specified. A wider range between the start and stop lines will produce a taller block as the cursor, e.g. the values 1, 15 cause the cursor to occupy the entire character block.

Any parameter may be omitted. Omitted parameters assume the current value.

Values outside the permitted ranges result in an "Illegal function call" error; previous values are retained.

Line 25 of the screen is initially used by the function key display. If you want to write to this line from a program, use KEY OFF to clear line 25, then use LOCATE 25,1:PRINT... to write to line 25.

Cursor blink is not selectable; the cursor always blinks 4 times per second.

```
10 LOCATE,,1
```

The above turns the cursor on.

```
10 LOCATE 1,1
20 LOCATE 5,5:PRINT "HELLO"
30 LOCATE ,,1
40 LOCATE ,,,12
```

Line 10 moves the cursor to the home position in the upper left corner. Line 20 prints "HELLO" at row 5, column 5. Line 30 makes the cursor visible with its position unchanged. In Line 40 the cursor's position and visibility are unchanged. The cursor will display at the bottom of the character, starting and ending on scan line 12.

LOF Function

FORMAT `v = LOF (filenum)`

PURPOSE Returns the length of a file in bytes.

■ ■ ■

filenum is the number used when the file was OPENed.

For random and sequential files, LOF returns the file size in bytes. For communications files, LOF returns the number of free bytes in the input buffer.

```
110 IF REC * RECSIZ > LOF(1)
    THEN PRINT "INVALID ENTRY"
```

In this example, the variables REC and RECSIZ contain the record number and record length, respectively. The calculation determines whether the specified record is beyond the end-of-file.

LOG Function

FORMAT v = LOG(x)

PURPOSE Returns the natural logarithm of x.

■ ■ ■

x must be greater than zero.

A natural logarithm is the logarithm to the base e.

```
PRINT LOG(45/7)
 1.860752
Ok
```

The logarithm of 45/7 is 1.860752.

LPOS Function

FORMAT **v** = LPOS(**n**)

PURPOSE Returns the current position of the print head in the printer's buffer.

■ ■ ■

n is the number assigned to the line printer (as in LPT**n**:).

This function does not necessarily give the physical position of the print head.

```
10 IF LPOS(0)>50 THEN LPRINT CHR$(13)
```

The above sends a carriage return character to the printer if the line length is more than 50 characters.

LPRINT and LPRINT USING Statements

FORMAT LPRINT [**list of expressions**][;]

LPRINT USING **string exp;list of expressions**[;]

PURPOSE Prints data on the printer (LPT1:).

■ ■ ■

list of expressions contains the string or numeric expressions that are to be printed. These expressions must be separated by semicolons (or commas if using the GWBASIC print zones).

string exp is the string constant or variable that identifies the format to be used for printing.

These statements are similar to PRINT and PRINT USING, except output goes to the printer.

See PRINT statement and PRINT USING statement for further explanation.

```
10 A=123
20 LPRINT "THIS IS OUTPUT ON THE PRINTER"
30 LPRINT USING "####"; A
RUN
```

The above prints the following on the printer:

```
THIS IS OUTPUT ON THE PRINTER
123
```

LSET and RSET Statements

FORMAT LSET **stringvar = x$**
RSET **stringvar = x$**

PURPOSE Moves data from memory to a random file buffer in preparation for a PUT statement, or left- or right-justifies a string in a given field.

■ ■ ■

stringvar is a variable defined in a FIELD statement.

x$ is a string expression.

If **x$** requires fewer bytes than specified in the FIELD statement for **stringvar**, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If **x$** 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 MKI$, MKS$, MKD$ functions.

See also Appendix A, Sequential and Random Files.

NOTE: These functions may also be used with a string variable not defined in a FIELD statement to justify the string in a given field. This is helpful when formatting printed output.

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

The above right-justifies the string N$ in a 20-character field.

```
150 LSET A$=MKS$(AMT)
```

This example changes AMT into character string A$ and left-justifies it so that there are no leading blanks.

MERGE Command

FORMAT MERGE **filespec**

PURPOSE Merges a program from a specified ASCII file into the program currently in memory.

■ ■ ■

filespec is a file specification (see Section 3, under "File and Device Information") in the form of a string constant, and identifies the file containing the program to be merged. The filespec consists of a filename or pathname with an optional device name. If used, the device name must be that of a diskette or hard drive. If the device name is omitted, the DOS default drive is used.

The program being merged must be in ASCII format or a "Bad file mode" error occurs. A program is in ASCII format if the A option was specified when the file was SAVEd.

If any lines in the disk file have the same numbers as in the program in memory, the lines from the file on disk will replace those in memory. (Merging may be thought of as "inserting" the program lines on disk into the program in memory.)

GWBASIC returns to command level after a MERGE command.

```
MERGE "A:NUMBRS"
```

In this example, the file "NUMBRS" on drive A: is merged with the program in memory.

MID$ Function

FORMAT **v$ = MID$(x$,n[,m])**

PURPOSE Returns a string of length **m** characters from **x$**, beginning with the **n**th character.

■ ■ ■

n and **m** must be in the range 1 to 255. If **m** is omitted or if there are fewer than **m** characters to the right of the **n**th character, all rightmost characters beginning with the **n**th character are returned. If **n** is greater than the number of characters in **x$** (i.e., if **n** > LEN(**x$**)), MID$ returns a null string.

See also the LEFT$ and RIGHT$ functions.

```
LIST
     10 A$="GOOD "
     20 B$="MORNING EVENING AFTERNOON"
     30 PRINT A$;MID$(B$,9,7)
     Ok
     RUN
      GOOD EVENING
     Ok
```

MID$ is used to select part of the string B$.

MID$ Statement

FORMAT MID$(**string-exp1,n[,m]**)=**string-exp2**

PURPOSE Replaces a portion of one string with another string.

■ ■ ■

n and **m** are integer expressions, and **string-exp1** and **string-exp2** are string expressions.

The characters in **string-exp1**, beginning at position **n**, are replaced by the characters in **string-exp2**. The optional **m** refers to the number of characters from **string-exp2** that will be used in the replacement. If **m** is omitted, all of **string-exp2** is used. However, regardless of whether **m** is omitted or included, the replacement of characters never goes beyond the original length of **string-exp1**.

See also MID$ function.

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
 KANSAS CITY, KS
```

The above shows two characters in string A$ being replaced by "KS".

MKDIR Command

FORMAT MKDIR **pathname**

PURPOSE Creates a new directory

■ ■ ■

pathname is a string expression of up to 63 characters and identifies the directory to be created. MKDIR works exactly like the DOS command MKDIR (see "Tree-Structured Directories" in Section 3).

Assuming that the current directory is the root, the statement:

```
MKDIR "SALES"
```

creates a directory named SALES under the root directory of the current drive (the format MKDIR "\SALES" would have the same effect since the root is the current directory).

```
MKDIR "\SALES\JOHN"
```

creates a subdirectory named JOHN under the SALES directory.

```
MKDIR "B:USERS
```

creates a sub-directory named USERS in the current directory on drive B.

See also the CHDIR and RMDIR commands.

MKI$, MKS$, MKD$ Functions

FORMAT
```
v$ = MKI$(integer expression)
v$ = MKS$(single-precision expression)
v$ = MKD$(double-precision expression)
```

PURPOSE Converts numeric values to string values.

▪ ▪ ▪

Any numeric value placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single-precision number to a 4-byte string. MKD$ converts a double-precision number to an 8-byte string.

Refer to CVI, CVS, CVD functions and to Appendix A, Sequential and Random Files.

```
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
    .
    .
```

Line 100 defines the fields in file 1. In line 110 AMT is converted to a string and put into the random file buffer. Line 120 puts a string in the buffer, and the following PUT statement writes data from the buffer to the file.

NAME Statement

FORMAT NAME **old-filespec** AS **new-filespec**

PURPOSE Renames a disk file.

■ ■ ■

old-filespec is the name of a file that already exists. **new-filespec** is what you now wish the file to be called. **filespec** is a file specification (see Section 3, under "File and Device Information") in the form of a string constant, and identifies the file. The filespec consists of a filename or pathname with an optional device name. If used, the device name must be that of a diskette or hard drive. If the device name is omitted, the DOS default drive is used.

After a NAME statement, the file exists on the same disk, in the same area of disk space, with the new name.

NAME cannot be used to rename directories. However, it can be used to move a file from the current directory to another on the same drive.

```
NAME "A:DRAFT" AS "PROPOSAL"
Ok
```

This renames "DRAFT" to "PROPOSAL" on drive A.

```
NAME "C:\X\CLIENTS" AS "\XYZ\P\CLIENTS"
```

This moves the file named CLIENTS from directory X on drive C to directory XYZ, subdirectory P on drive C.

NEW Command

FORMAT NEW

PURPOSE Deletes the program currently in memory and clears all variables.

■ ■ ■

This command clears memory before entering a new program, and also closes all files and turns tracing off. You can only use NEW in direct mode. GWBASIC returns to command level after NEW is executed.

See also TRON and TROFF.

NEW

The program in memory will be deleted.

OCT$ Function

FORMAT **v$** = OCT$(**n**)

PURPOSE Returns the octal value of a decimal.

■ ■ ■

n is a decimal value in the range -32768 to 65535.

See also HEX$ function for hexadecimal conversion.

```
PRINT OCT$(100)
 144
Ok
```

Decimal 100 is octal 144.

ON COM(n) Statement

FORMAT ON COM(**n**) GOSUB **line-number**

PURPOSE Specifies the first line number of a subroutine to be performed when activity occurs on a communications channel.

■ ■ ■

line-number is the number of the first line of a subroutine.

n is the number of the communications channel, a number in the range 1 to 4.

ON COM will only be executed if a COM(**n**) ON statement has been executed. If event trapping is enabled, and if **line number** in the ON COM statement is not zero, GWBASIC checks between statements to see if communications activity has occurred on the specified channel. If communications activity has occurred, a GOSUB will be performed to the specified line.

If a COM OFF statement has been executed for the communications channel, the GOSUB is not performed and is not remembered.

If a COM STOP statement has been executed for the communications channel, the GOSUB is not performed, but will be performed as soon as a COM ON statement is executed.

A **line-number** of zero disables the communications trap.

When an event trap occurs (i.e., the GOSUB is performed), an automatic COM STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a COM ON statement unless an explicit COM OFF was performed inside the subroutine.

RETURN **line-number** may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping does not take place when GWBASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

ON ERROR GOTO Statement

FORMAT ON ERROR GOTO `line`

PURPOSE Enables errors to be trapped and specifies the first line of the error-handling subroutine.

■ ■ ■

`line` is the number of the first line of the error-handling subroutine. If `n` does not exist, an "Undefined line number" error occurs.

Once error handling has been enabled, all errors detected, including direct mode errors (e.g., syntax errors), will cause a jump to the specified error-handling routine.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution.

An ON ERROR GOTO 0 statement that appears in an error-trapping subroutine causes GWBASIC to stop and print the error message for the error that caused the trap. It is recommended that all error-handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE: If an error occurs during execution of an error-handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error-handling routine.

Use RESUME to exit from an error-trapping routine.

```
10 ON ERROR GOTO 1000
      .
      .
      .
1000 PRINT "Error";ERR;"at line";ERL
      .
      .
      .
```

These are sample statements that would be used in error handling.

ON...GOSUB and ON...GOTO Statements

FORMAT ON **n** GOTO **line[,line]...**

ON **n** GOSUB **line[,line]...**

PURPOSE Branches to one of several specified line numbers, depending on a specified value.

■ ■ ■

n is a number in the range 0 to 255. If **n** is negative or greater than 255, an "Illegal function call" error occurs. If **n** is zero or greater than the number of items in the list (but less than or equal to 255), GWBASIC continues with the next executable statement.

The value of **n** determines which line number in the list will be used for branching. For example, if **n** is three, the third line number in the list will be the destination of the branch.

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

```
100 ON L-1 GOTO 150,300,320,390
```

In the above, if L-1 equals 0 or is greater than 4, the program will continue with the next statement.

If L-1 equals 1, the program branches to line 150.
If L-1 equals 2, the program branches to line 300.
If L-1 equals 3, the program branches to line 320.
If L-1 equals 4, the program branches to line 390.

ON KEY(n) Statement

FORMAT ON KEY(**n**) GOSUB **line number**

PURPOSE Specifies the first line number of a subroutine to be performed when a specified function key, cursor direction key or user-defined key is pressed.

■ ■ ■

n is the number of the key to be trapped:

1-10	Function keys F1 to F10
11	Cursor up
12	Cursor left
13	Cursor right
14	Cursor down
15-20	User-defined keys (see the KEY statement)

line number is the number of the first line of a subroutine that is to be performed when the specified function or cursor direction key is pressed.

A **line number** of zero disables the event trap.

The ON KEY statement will only be executed if a KEY(**n**) ON statement has been executed to enable event trapping. If event trapping is enabled, and if the line number in the ON KEY statement is not zero, GWBASIC checks between statements to see if the specified function key, cursor direction key or user-defined key has been pressed. If so, a GOSUB will be performed to the specified line.

If a KEY(**n**) OFF statement has been executed for the specified key, the GOSUB is not performed and is not remembered.

If a KEY STOP statement has been executed for the specified key, the GOSUB is not performed, but will be performed as soon as a KEY(**n**) ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic KEY(**n**) STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a KEY(**n**) ON statement unless an explicit KEY(**n**) OFF was performed inside the subroutine.

RETURN **line number** may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping does not take place when GWBASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

NOTE: When a key is trapped, that occurrence of the key is destroyed. Therefore, you cannot subsequently use the INPUT or INKEY$ statement to find out which key caused the trap. So if you want to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

```
10 KEY 4,TRON            'assigns function key 4
20 KEY(4) ON             'enables event trapping
    .
    .
    .
70 ON KEY(4) GOSUB 200
    .
    .
    .
(Function key 4 pressed)
    .
    .
    .
200                      'Subroutine for TRON
```

The above is a subroutine for function key 4.

```
100 KEY 15, CHR$(&H04) + CHR$(83)
105 REM ** Key 15 is now Control-S **
110 KEY(15) ON
     .
     .
     .
1000 PRINT "If you want to stop processing"
1010 PRINT "for a break press the Ctrl key"
1020 PRINT "and the S at the same time".
```

```
1030 ON KEY(15) GOSUB 3000
      .
      .
(operator presses Ctrl-S)
      .
      .
3000 REM ** Suspend processing loop
3010 CLOSE #1
3020 RESET
3030 CLS
3035 PRINT "Enter CONT to continue"
3040 STOP
3050 OPEN "A", #1, "ACCOUNTS.DAT"
3060 RETURN
```

In the above example, Ctrl-S has been enabled to enter a subroutine which closes the files and stops program execution until the operator is ready to continue.

ON PEN Statement

FORMAT ON PEN GOSUB **line number**

PURPOSE Specifies the first line number of a subroutine to be performed when the light pen is activated.

▩ ▩ ▩

line number is the first line of a subroutine.

If **line number** is zero, the trap is disabled.

The ON PEN statement will only be executed if a PEN ON statement has been executed. If event trapping is enabled, and if **line number** is not zero, GWBASIC checks between statements to see if the pen has been activated. If it has, a GOSUB will be performed to the specified line.

If a PEN OFF statement has been executed, the GOSUB is not performed and is not remembered.

If a PEN STOP statement has been executed, the GOSUB is not performed, but will be performed as soon as a PEN ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic PEN STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a PEN ON statement unless an explicit PEN OFF was performed inside the subroutine.

RETURN **line number** may be used to return to a specific line number from the trapping subroutine. Use with care, however, because any other GOSUBs, WHILEs, or FORs active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping does not take place when GWBASIC is not executing a program, and trapping is automatically disabled when an error trap occurs.

See PEN statement and function in this section for an example of the use of the light pen.

ON PLAY(n) Statement

FORMAT ON PLAY(**n**) GOSUB **line number**

PURPOSE Branches to a specified subroutine when the music queue contains fewer than **n** notes. This permits continuous music during program execution.

■ ■ ■

n is an integer expression in the range 1 through 32.

line number is the starting line number of a subroutine to which control is passed when the event occurs (i.e. when the queue contains fewer than **n** notes).

The ON PLAY statement will only be executed if a PLAY ON statement has been executed to enable event trapping. If trapping is enabled, and **line number** is not 0, GWBASIC checks between statements to see if the music queue contains fewer than **n** notes. If so, a GOSUB is performed to the specified line.

If a PLAY OFF statement has been executed the GOSUB is not performed and is not remembered.

If a PLAY STOP statement has been executed the GOSUB is not performed, but will be performed as soon as a PLAY ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic PLAY STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a PLAY ON statement unless an explicit PLAY OFF was performed inside the subroutine.

RETURN **line number** returns to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

A play event trap is issued only when playing background music (i.e., PLAY "MB..." is active). Play event traps are not issued when PLAY is executing in music foreground mode (i.e., PLAY "MF...", the default value).

A play event trap is not issued if the background music queue is already empty when PLAY ON is executed.

Choose low values for **n**. An ON PLAY(32) statement will cause event traps so often that there will be little time to execute the rest of the program.

See also the PLAY ON, PLAY OFF and PLAY STOP statements.

In this 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 STRIG(n) Statement

FORMAT ON STRIG(**n**) GOSUB **line number**

PURPOSE Specifies the first line number of a subroutine to be performed when the joystick trigger is pressed.

■ ■ ■

n is the number of the joystick trigger.

line number is the first line of a subroutine. If a line number is zero, the trap is disabled.

The ON STRIG statement will only be executed if a STRIG ON statement has been executed. If event trapping is enabled, and if **line number** is not zero, GWBASIC checks between statements to see if the joystick trigger has been pressed. If it has, a GOSUB will be performed to the specified line.

If a STRIG OFF statement has been executed, the GOSUB is not performed and is not remembered.

If a STRIG STOP statement has been executed, the GOSUB is not performed, but will be performed as soon as a STRIG ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic STRIG STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a STRIG ON statement unless an explicit STRIG OFF was performed inside the subroutine.

RETURN **line number** may be used to return to a specific line number from the trapping subroutine. Use with care, however, because any other GOSUBs, WHILEs, or FORs active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping does not occur when GWBASIC is not executing a program, and trapping is disabled automatically when an error trap occurs.

See the STRIG statement and function in this section for a joystick example.

ON TIMER(n) Statement

FORMAT ON TIMER(**n**) GOSUB **line number**

PURPOSE Branches to a subroutine when a specified time interval has elapsed.

■ ■ ■

n is a numeric expression in the range 1 to 86400 (1 second to 24 hours). Values outside this range generate an "Illegal function call" error.

line number is the starting line number of a subroutine to which control is passed when the event occurs (i.e., when the specified number of seconds has elapsed).

ON TIMER causes an event trap every **n** seconds. The ON TIMER statement will only be executed if a TIMER ON statement has been executed to enable event trapping. If event trapping is enabled, and if the line number in the ON TIMER statement is not zero, GWBASIC checks between statements to see if the time has been reached. If it has, a GOSUB will be 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 (i.e., the GOSUB is performed), an automatic TIMER STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a TIMER ON statement unless an explicit TIMER OFF was performed inside the subroutine.

RETURN **line number** may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

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 TIME$;
10030 LOCATE OLDROW,OLDCOL 'Restore row & col
10040 RETURN
```

See also the TIMER ON, TIMER OFF and TIMER STOP statements.

OPEN Statement

FORMAT OPEN **"mode1",[#]file number,filespec[, record length]**

OPEN **filespec**[FOR **mode2**] AS **[#]file number** [LEN=**record length**]

PURPOSE Allows input or output (I/O) to a data file (sequential or random) or device. Files and devices must be opened before any I/O operation is performed on them.

■ ■ ■

A file is opened for output in order to write data to it, and opened for input if data is to be read from it.

mode1 is a string expression whose first character is one of the following:

O specifies sequential output mode

I specifies sequential input mode

R specifies random input/output mode

A specifies sequential output mode and sets the file pointer at the end of the file and the record number as the last record of the file. A PRINT# or WRITE# statement will then extend (append) the file.

Note that the value of **mode1** must be enclosed in quotes.

mode2 is one of the following:

OUTPUT specifies sequential output mode

INPUT specifies sequential input mode

APPEND specifies sequential output mode and sets the file pointer at the end of the file and the record number as the last record of the file. A PRINT# or WRITE# statement will then extend (append) the file.

If **mode2** is omitted, random access mode is assumed. Note that this mode cannot be expressed explicitly for **mode2**.

file number is an integer between 1 and 15. The number is associated with the file for as long as it is OPEN. The number is a shorthand notation for the file. It is also used to refer other disk I/O statements to the file.

filespec is a file specification (see Section 3, under "File and Device Information") in the form of a string constant, and identifies the file or device to be opened. The filespec consists of a filename or pathname with an optional device name (which must be that of a diskette or hard drive), or the name of a user-installed device driver. Valid device names are KYBD:, SCRN:, COMn:, LPTn: and CON:. If the device name is omitted, the DOS default drive is used.

record length is an integer that, if included, sets the record length for random files. This option will be ignored if used with sequential files.

record length cannot exceed the maximum set with the /S: option of the GWBASIC command. If the **record length** option is not used, the default length is 128 bytes, unless the /I and /R options of the GWBASIC command have been used.

A disk file must be opened before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer.

Opening a file in O or OUTPUT mode causes data in the file to be overwritten. Opening a file in A or APPEND mode preserves the original file contents and adds new data at the end.

NOTE: A file can be opened for sequential input or random access on more than one file number at a time. A file may be opened for output, however, on only one file number at a time.

If you are using tree-structured directories (see Section 3), you cannot open a file for sequential output or append if a file with the same name is already open in any mode on the same disk, even if the two are completely different files.

```
10 OPEN "I",2,"INVEN"
```

Opens the file "INVEN" for sequential input.

```
10 OPEN "MAILING.DAT" FOR APPEND AS 1
```

Opens the file "MAILING.DAT" for extension.

```
10 OPEN "\DEV\PRT" FOR OUTPUT AS #1
```

If a user-installed device driver named PRT is used for the printer, this statement will open the printer for sequential output.

```
10 OPEN "LPT:" FOR OUTPUT AS #1
```

Opens the printer for output using the GWBASIC device driver.

```
10 OPEN "\DEV\LPT1" FOR OUTPUT AS #1
```

Opens the printer for output using the DOS device driver.

OPEN "COM..." Statement

FORMAT OPEN "COMn:**[speed][,[parity][,[data]**
[,[stop][,RS][,CS[n]][,DS[n]][,CD[n]]
[,BIN][,ASC][,LF]]]]"[FOR **mode**] AS
[#]filenum [LEN=**record length**]

PURPOSE Opens and initializes a communications channel for input/output.

■ ■ ■

OPEN "COM..." must be specified before a device connected to the specified channel is used for RS-232 communications. The channel is treated as a file with a file number, and is referenced as such in subsequent I/O statements.

NOTE: Although default values can be used for the parameters, you are strongly recommended to define the parameter values explicitly in each case.

n is the number of the channel to be opened, an integer in the range 1 to 4. The number 1 refers to the built-in asynchronous communications adapter (serial port), while 2 through 4 can be used for add-on adapters.

speed is the baud rate, in bits per second, of the device connected to the channel to be opened. Valid speeds are 75, 110, 150, 300, 600, 1200, 2400, 4800 and 9600. Default is 300 bps.

parity indicates the type of parity for transmission and the type of parity checking to be performed on data received. Valid entries are:

N - none

E - even parity (default)

O - odd parity

S - space (parity bit transmitted and received as a 0 bit)

M - mark (parity bit transmitted and received as a 1 bit)

data indicates the number of bits per byte. Valid entries are 7 (default) or 8.

stop indicates the number of stop bits. Valid entries are 1 (default) or 2.

RS suppresses the RTS (Request To Send) signal. The RTS line is activated when OPEN "COM..." is executed unless this parameter is included.

CS[**n**] controls the CTS (Clear To Send) signal.

DS[**n**] controls the DSR (Data Set Ready) signal.

CD[**n**] controls the CD (Carrier Detect) signal, also known as the RLSD (Received Line Signal Detect) signal.

n for the CS, DS and CD parameters specifies the wait time for the corresponding signal in milliseconds, in the range 0 to 65535. If the specified signal is not received after **n** milliseconds, the message "Device timeout" is displayed. If **n** is 0, no line checking is performed. Default for CS and DS is 1000. If CD is omitted, that line status is not checked.

A "Device timeout" error will normally occur if GWBASIC attempts to execute I/O statements to a communications file and the CTS or DSR lines are not active. With CS and DS you can either bypass the normal checking of those signals by giving a value of 0, or you can specify a different wait time for the test.

BIN opens the device in binary mode. BIN is selected by default unless ASC is specified. In the BIN mode, tabs are not expanded to spaces, a carriage return is not forced at the end-of-line, and Ctrl-Z is not treated as end-of-file. When the channel is closed, Ctrl-Z is not sent over the RS-232 line. BIN supersedes the LF option.

ASC opens the device in ASCII mode. In this mode, tabs are expanded, carriage returns are forced at the end-of-line, Ctrl-Z is treated as end-of-file, and XON/XOFF is enabled. When the channel is closed, Ctrl-Z is sent over the RS-232 line.

LF specifies that a line feed is to be sent after a carriage return, allowing communications files to be printed on a serial printer. When LF is specified, a line-feed character (hex 0A) is automatically sent after each carriage return character (hex 0D), including the carriage return sent as a result of the width setting. Note that INPUT# and LINE INPUT#, when used to read from a COM file that was opened with the LF option, stop when they see a carriage return, ignoring the line feed.

mode is the mode in which the file referenced by **filenum** is opened and is one of the following:

OUTPUT specifies sequential output mode

INPUT specifies sequential input mode

If **mode** is omitted, random access mode is assumed. Note that this mode cannot be explicitly expressed for this parameter.

filenum is a file number; the communications channel is treated as a file with the name COM**n**, and the file is referred to by **filenum** in other communications I/O statements.

record length is the maximum number of bytes that can be read from the communications buffer when using a GET or PUT statement. Default value is 128.

Any format errors in this statement will result in a "Bad file name" error. The incorrect parameter will not be shown.

The **speed, parity, data**, and **stop** options must be listed in the order shown (for the last three of these options, include the preceding commas even if you omit the values). The remaining options may be listed in any order after the first four options.

```
10 OPEN "COM1:9600,N,8,1,BIN" AS #2
```

Opens communications channel 1 at a speed of 9600 baud with no parity bit, 8 data bits, and 1 stop bit. Input/output will be in the binary mode. Other lines in the program may now access channel 1 as device number 2.

OPTION BASE Statement

FORMAT OPTION BASE **n**

PURPOSE Declares the minimum value for array subscripts.

■ ■ ■

n is 1 or 0.

The default base is 0. If the statement:

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is 1.

If used, OPTION BASE must be coded before you define or use any arrays.

Chained programs may have an OPTION BASE statement if no arrays are passed between them or the specified base is identical in the chained programs. A chained program will inherit the OPTION BASE value of the chaining program.

OUT Statement

FORMAT OUT `i,j`

PURPOSE Sends a byte to a machine output port.

■ ■ ■

`i` is the port number, an integer in the range 0 to 1023 (&H3FF).

`j` is the data to be transmitted, an integer in the range 0 to 255.

For examples of using the OUT statement with a communications program, see Section 6 under "Accessing the Registers".

A port address map is provided in Table 7-6.

```
100 OUT 12345,255
```

The value 255 is sent to output port 12345.

PAINT Statement

FORMAT PAINT **(x,y)[,paint[,border][,background]]**

PURPOSE Fills a graphics area with the color or pattern specified.

■ ■ ■

This statement can only be used in graphics mode (SCREEN 1, 2, 104 or 105).

x and **y** are the coordinates where painting is to begin. Painting should always start on a non-border point. If painting starts inside a bordered figure, the figure is painted up to the border. If painting starts outside a bordered figure, the background is painted. You can use the STEP option to specify these coordinates relative to the last point referenced - see the CIRCLE or LINE statements for details.

If **paint** is a numeric expression, the figure will be painted with the color corresponding to that number (see the COLOR statement). In medium resolution (SCREEN 1), **paint** ranges from 0 to 3, where 0 indicates the background color and 1 to 3 denote colors from the current palette (see Table 7-2). In high resolution (SCREEN 2) and super resolution (SCREEN 104 and 105), **paint** can be either 0 (background color) or 1 (foreground color). Default is 3 for medium resolution and 1 for high- and super resolution.

If **paint** is a string expression, PAINT will execute "tiling" (see below) to paint the figure with a pattern instead of a single color.

border identifies the border color of the figure to be filled. When the border color is encountered, painting of the current line will stop. If **border** is not specified, the value of **paint** will be used.

background is a string expression used in tiling (see below). The default is CHR$(0).

PAINT can be used to fill any figure, but painting complex figures may result in an "Out of Memory" error. If this happens, use the CLEAR statement to increase the amount of stack space available.

PAINT permits coordinates outside the screen or viewport. No error message is issued if this occurs.

Tiling

Tiling allows you to design a pattern with which to paint a figure. You specify the pattern by the value of the **paint** parameter, which in this case must be a string expression. The pattern design, or "tile", is 8 bits wide (corresponding to the width of one character on the screen) and can be up to 64 bytes long (each byte corresponding to one screen line).

To specify a design, 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$(**n**) up to 64 will generate an image of the bit arrangement of **n**. For example, the decimal number 85 is binary "01010101"; the graphic image line on a monochrome screen generated by CHR$(85) is an eight-point line, with even-numbered points turned white, and odd ones black.

A monochrome screen can be painted with a pattern of Xs with the following statements:

```
10 CLS : SCREEN 105
20 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 →

x,y	bit of tile byte 7	6	5	4	3	2	1	0		Tile Byte
0,0	x							x	CHR$(129)	1
0,1		x					x		CHR$(66)	2
0,2			x			x			CHR$(36)	3
0,3				x	x				CHR$(24)	4
0,4				x	x				CHR$(24)	5
0,5			x			x			CHR$(36)	6
0,6		x					x		CHR$(66)	7
0,7	x							x	CHR$(129)	8

The pattern is repeated uniformly over the area to be painted. Each byte of the string is rotated as required to align the pattern along the y-axis.

Occasionally you may want to tile paint over an already-painted area that is the same color or pattern as two consecutive lines in the tile pattern. Normally, PAINT terminates when it encounters two consecutive lines of the same color or pattern as the point being set (i.e. the point is surrounded).

Use the **background** parameter to avoid this condition. For example, if you want to draw alternating blue and red lines on a red background, painting would normally stop at the first attempt to draw a red line over the background. However, if you give **background** the value CHR$(&HAA), which corresponds to red if the current palette is 0 (see list below), this tells the program that the background is red and allows the red line to be drawn over it. The values that can be given for **background** and the colors they correspond to are:

Value	Palette 0	Pallet 1
CHR$(&H55)	green	cyan
CHR$(&HAA)	red	magenta
CHR$(&HFF)	brown	white

You cannot specify more than two consecutive bytes in the tile pattern that match **background**. Specifying more than two will result in an "Illegal function call" error.

Some examples follow of the use of the PAINT statement:

```
5 SCREEN 1:CLS
10 PAINT (5,15),2,0
```

begins painting at coordinates 5,15 with color 2 and border color 0, and fills to a border.

```
10 CLS:CIRCLE (160,100),100
20 PAINT (160,100),CHR$(129) + CHR$(66) +
   CHR$(36) + CHR$(24) + CHR$(24) + CHR$(36) +
   CHR$(66) + CHR$(129)
```

draws a circle and fills it with the pattern of Xs described above.

PEEK Function

FORMAT **v** = PEEK(**n**)

PURPOSE Returns the byte read from memory location **n**.

■ ■ ■

The returned value is an integer in the range 0 to 255. **n** is a numeric expression and is the offset from the current segment, which was defined by the last DEF SEG statement. **n** must be in the range -32768 to 65535.

See POKE, the complementary function of this statement.

```
X = PEEK (1327)
PRINT X
 199
```

The above displays the value at memory location 1327.

```
A = PEEK(&H5A00)
```

In this example, the value at the location with hexadecimal address 5A00 is loaded into the variable A.

PEN Statement and Function

FORMAT
```
PEN ON
PEN OFF
PEN STOP
x = PEN(n)
```

PURPOSE Reads the light pen and enables, disables, or stops trapping the pen.

■ ■ ■

x is the numeric variable receiving the PEN value.

n is a number from 0 to 9. This function traps downstrokes of the light pen by reading the following parameters of **n**:

0 Indicates whether pen was down since last poll. Returns -1 if down, 0 if not.

1 Returns the x-coordinate of the point where the pen was last pressed.

2 Returns the y-coordinate of the point where the pen was last pressed.

3 Returns the current pen switch value, -1 if down, 0 if up.

4 Returns the last known valid x-coordinate.

5 Returns the last known valid y-coordinate.

6 Returns the character row position where pen was last pressed.

7 Returns the character column position where pen was last pressed.

8 Returns the last known character row where the pen was positioned.

9 Returns the last known character column where the pen was positioned.

PEN ON enables both the read function and event trapping.

PEN STOP disables both the light pen read function and event trapping but remembers a PEN event so that it can be trapped as soon as there is a PEN ON.

PEN OFF not only disables both the read function and event trapping, it also does not remember subsequent activity.

The initial setting is OFF. A PEN ON statement must be executed before any pen read function calls can be made. If the function is called when the setting is OFF, an "Illegal function call" error will result.

PEN ON also enables event trapping by an ON PEN statement (see ON PEN statement in this section). While trapping is enabled, and if a nonzero line number is specified in the ON PEN statement, GWBASIC checks between every statement to see if the light pen has been activated. If it has, the program transfers to the ON PEN statement.

The pen should not be used in the border area of the screen. Values returned from that area will be inaccurate.

To speed program execution, use PEN OFF for programs not using the light pen.

```
5 CLS
10 PEN ON
20 P=PEN(3)
30 LOCATE 1,1 : PRINT "STATUS IS";
40 IF P THEN PRINT "DOWN" ELSE PRINT "UP"
50 GOTO 20
```

The above produces an endless loop to print the current pen switch status (UP/DOWN).

PLAY Statement

FORMAT PLAY **string**

PURPOSE Plays music.

■ ■ ■

string is an expression consisting of single character commands as follows:

On Sets the current octave. There are 7 octaves, numbered 0 (lowest) to 6, each beginning with C and ending with B. Middle C begins octave 3. The default octave is 4.

> Increments the octave by one if placed before a note or series of notes. >> increments by two octaves, etc. The octave will not go higher than 6.

< Decrements the octave by one if placed before a note or series of notes. << decrements by two octaves, etc. The octave will not go lower than 0.

A to G with optional #, +, or -
Plays the indicated note in the current octave. The note followed by a number or plus sign indicates a sharp. Followed by a minus, it indicates a flat. The #, +, or - must correspond to a black key on a piano (for example, E+ is invalid).

Nn A note numbered 0 to 84 (in the seven possible octaves there are 84 notes). Zero indicates a rest. This is an alternative form to **On** and A-G.

Ln Sets the length of the notes to follow; **n** is in the range 1 to 64. L1 is a whole note (semibreve), L4 is a quarter note (crotchet), etc. Intermediate values can be used, for example L3 can be used for a triplet (three notes in the time of two) of half notes, L6 for a triplet of quarter notes, and so on.

The length may also follow the note when, for example, you want to change it only for that note. Thus A16 is equivalent to L16A.

Pn Rest (pause). The range of **n** may be from 1 to 64, indicating the length of the rest in the same way as for L.

A dot or period after a note causes it to be played as a dotted note (its length is multiplied by 3/2). More than one dot is allowed, which causes the length of the note to be increased by the appropriate 3/2 multiple. Thus "C.." plays 9/4 as long as its length indicates, and "C..." plays 27/8 as long. Periods may also appear after a rest (P) to lengthen it in the same way.

Tn The tempo sets the number of quarter notes in a minute in the range 32 to 255. The SOUND statement lists common tempos and equivalent beats per minute. The default is 120.

MF Music foreground. Each sound begins only after the previous sound is finished. This is the default. It is also valid for SOUND.

MB Music background. Each sound is written to a buffer, which allows a program to continue executing while music plays in the background. The maximum number of notes (or rests) that can be played in background at a time is 32. This option is also valid for SOUND.

MN Music normal (nonlegato). This is the default setting when neither ML nor MS is specified. Each note is played at 7/8L, creating the nonlegato sound.

ML Music legato. Each note plays the full length set with L, which creates "connected" notes.

MS Music staccato. Each note plays at 3/4L to create a strong disconnected sound.

X a$; Executes the specified string. You can use this feature to store a frequently-repeated subtune in a string variable and call it from the PLAY statement.

Note that because of the slow clock interrupt rate, some notes will not play at high tempos, e.g., L64 at T255. You may discover other combinations as you experiment with the possibilities of this statement.

```
10 PLAY "ms o3 l16 g l8 e. l16 e l12 ee-e o4
   l8 c. o3 l6 g l8 g. l16 e l8 f. l16 f l8
   f. l16 g l4 a"
```

The above plays part of a march tune well-known on both sides of the Atlantic.

PLAY(n) Function

FORMAT **v** = PLAY(**n**)

PURPOSE Returns the number of notes currently in the background music queue.

■ ■ ■

(**n**) is a dummy argument and may be any value.

PLAY(**n**) only returns a value if PLAY is currently executing in music background (MB) mode. It returns 0 if PLAY is currently in music foreground (MF) mode, which is the default.

PLAY ON, PLAY OFF, PLAY STOP Statements

FORMAT
PLAY ON
PLAY OFF
PLAY STOP

PURPOSE Enables (ON), disables (OFF) or suspends (STOP) music event trapping (i.e. testing whether the music queue has the number of notes specified in an ON PLAY(**n**) statement).

■ ■ ■

ON PLAY(**n**) must first be set with a line number before BASIC checks whether a music event has occurred. If so, the program will perform a GOSUB to the trap routine at the line number specified in ON PLAY(**n**).

PLAY STOP suspends trapping, but if a subsequent PLAY ON statement is executed the GOSUB is performed immediately.

PLAY OFF stops trapping altogether. If a subsequent PLAY ON statement is executed the GOSUB is not performed.

PMAP Function

FORMAT **v = PMAP (coord, function)**

PURPOSE Maps world coordinates created by the WINDOW statement to to physical locations, or maps physical locations to world coordinates.

■ ■ ■

This statement can only be used in graphics mode (SCREEN 1, 2, 104 or 105).

coord is the coordinate of the point to be mapped.

function is one of the following:

0 Maps world coordinate to physical x-coordinate.

1 Maps world coordinate to physical y-coordinate.

2 Maps physical coordinate to world x-coordinate.

3 Maps physical coordinate to world y-coordinate.

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.

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 coordinate is (200,200). The screen coordinates are (0,0) in the upper left-hand corner and (639,399) in the lower right for 400-line systems, and 639,324 for 325-line systems. Then:

```
X = PMAP(80,0)
```

would return the screen x-coordinate of the window x-coordinate 80:

```
0
```

The PMAP function in the statement:

```
Y = PMAP(200,1)
```

would return the screen y-coordinate of the window y-coordinate 200:

```
399 (or 324)
```

The PMAP function in the statement:

```
X = PMAP(619,2)
```

would return 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)
```

would return the "world" y-coordinate that corresponds to the screen or viewport y-coordinate 100:

```
140
```

POINT Function

FORMAT **v** = POINT (**x-coordinate,y-coordinate**)
v = POINT (**n**)

PURPOSE Reads the color value of a point on the screen, or returns the current graphics cursor coordinates.

This function can only be used in graphics mode (SCREEN 1, 2, 104 or 105).

x-coordinate and **y-coordinate** are the coordinates of the screen point for which the color is to be read.

If the specified point is out of range, the value -1 is returned.

n is a number from 0 to 3, as follows:

0 returns the current physical x-coordinate (the coordinate on the screen or current viewport)

1 returns the current physical y-coordinate

2 returns the current logical x-coordinate. If the WINDOW statement is not active, this will be the same as for 0 above.

3 returns the current logical y-coordinate. If the WINDOW statement is not active, this will be the same as for 1 above.

```
5 SCREEN 105
10 IF POINT(I,I)<>0 THEN PRESET (I,I)
   ELSE PSET (I,I)
```

The above is one way to invert the color of point (**I,I**) on a monochrome screen.

```
10 PSET (I,I),1-POINT(I,I)
```

This version of line 10 has exactly the same effect.

POKE Statement

FORMAT POKE **i,j**

PURPOSE Writes a byte into a memory location.

■ ■ ■

i and **j** are integer expressions.

i is the memory address, which must be in the range -32768 to 65535. **i** is the offset from the current segment, which was set by the last DEF SEG statement. (See DEF SEG statement.) Note that, if you specify a negative value for **i**, GWBASIC will interpret it as described in the specification of the VARPTR function.

j is the data byte, and must be in the range 0 to 255.

CAUTION

Use this statement carefully. It will cause severe problems if used incorrectly.

The complementary function to POKE is PEEK. (See PEEK function.)

```
10 POKE &H5A00,&HFF
```

The above puts the FF hex into 5A00 hex in the current segment.

POS Function

FORMAT **v = POS(n)**

PURPOSE Returns the current column position of the cursor.

■ ■ ■

n is a dummy argument, and can be any value.

CSRLIN finds the row position of the cursor.

See also LPOS function.

```
IF POS(0)>50 THEN BEEP
```

A beep is emitted if the cursor is beyond position 50 on the screen.

PRESET Statement

FORMAT PRESET [STEP](**x-coordinate,y-coordinate**) [,**color**]

PURPOSE Draws a specified point on the screen (default color is the background color).

■ ■ ■

This statement can only be used in graphics mode (SCREEN 1, 2, 104 or 105).

x-coordinate and **y-coordinate** specify the point that is to be set.

color is the number of the color to be used for the specified point. In medium resolution (SCREEN 1), **color** ranges from 0 to 3, where 0 indicates the background color and 1 to 3 denote colors from the current palette (see Table 7-2). In high resolution (SCREEN 2) and super resolution (SCREEN 104 and 105), **color** can be either 0 (background color) or 1 (foreground color). Default for all modes is the background color.

The STEP option, if used, means that **x-coordinate** and **y-coordinate** are relative to the most recent cursor location. For example, if the most recent location was (10,10), then STEP (10,5) would reference the point at (20,15).

PRESET works exactly like PSET except that if **color** is not specified, the background color is selected (default color for PSET is the foreground color).

If an out-of-range coordinate is given, no action is taken, nor is an error message given.

```
1 SCREEN 105
5 REM DRAW A LINE FROM (0,0) TO (100,100)
10 FOR I=0 TO 100
20 PRESET (I,I),1
30 NEXT
35 REM NOW ERASE THAT LINE
40 FOR I=0 TO 100
50 PRESET STEP (-1,-1)
60 NEXT
```

The above draws a line from (0,0) to (100,100) and then erases it by overwriting it with the background color.

PRINT Statement

FORMAT PRINT **[list of expressions[{, | ;}]]**

PURPOSE Outputs data on the screen.

■ ■ ■

If **list of expressions** is omitted, a blank line is printed. If **list of expressions** is included, the values of the expressions are displayed on the screen. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

You can use a question mark (?) instead of the word PRINT as a form of shorthand. It will be interpreted as PRINT, and will appear as PRINT in subsequent listings.

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. GWBASIC divides the line into print zones of 14 spaces each. Within the list of expressions, punctuation has the following effect:

,	causes next value to be printed at start of next zone
;	causes next value to be printed immediately after last value
spaces	have the same effect as semicolons (;)

If a comma or semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing in the same way as above. If the list of expressions terminates without a comma or a semicolon, a carriage return is performed at the end of the line. If the printed line is longer than the screen width, GWBASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single-precision numbers that can be represented with six or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 1E-7 is output as .0000001, and 1E-8 is output as 1E-08.

Double-precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 1D-15 is output as .0000000000000001, and 1D-16 is output as 1D-16.

See also LPRINT and LPRINT USING statements.

```
10 X=5
20 PRINT X+5, X-5, X*(-5),X^5
30 END
RUN
 10           0            -25                  3125
Ok
```

The commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

```
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

?
```

The semicolon at the end of line 20 causes both PRINT statements to be printed on the same line. Line 40 causes a blank line to be printed before the next prompt.

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

The semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Remember that a number is always followed by a space, and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

PRINT USING Statement

FORMAT PRINT USING **v$;list of expressions [;]**

PURPOSE Prints strings or numbers using a specified format.

■ ■ ■

v$ is a string constant or variable consisting of special formatting characters that determine the field and the format of the printed strings or numbers. See String and Numeric fields below.

list of expressions contains the string or numeric expressions that are to be printed, separated by semicolons.

String Fields

When printing strings with PRINT USING, one of three characters may be used to format the string field:

! specifies that only the first character in the given string is to be printed.

"\n spaces\"
specifies that 2+n characters from the string are to be printed (i.e., 2 plus the number of spaces). If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on.

If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

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   !!
```

& specifies a variable-length string field. When the field is specified with &, the string is printed without modification.

Example:

```
10 A$="LOOK": B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
 LOUT
```

Numeric Fields

When printing numbers with PRINT USING, the following special characters may be used to format the numeric field:

represents each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

(decimal point) may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

Examples:

```
PRINT USING "##.##";.78
 0.78
```

```
PRINT USING "###.##";987.654
 987.65

PRINT USING "##.##    ";10.2,5.3,66.789,.234
 10.20     5.30    66.79     0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

- at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

Examples:

```
PRINT USING "+##.##    ";-68.95,2.4,55.6,-.9
 -68.95     +2.40    +55.60     -0.90

PRINT USING "##.##-    ";-68.95,22.449,-7.01
 68.95-    22.45       7.01-
```

** at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. ** also specifies positions for two more digits.

Example:

```
PRINT USING "**#.# ";12.39,-0.9,765.1
 *12.4     *-0.9     765.1
```

$$ causes a dollar sign to be printed to the immediate left of the formatted number. $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

Example:

```
PRINT USING "$$###.##";456.78
 $456.78
```

**$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign. The exponential format cannot be used with **$. When negative numbers are printed, the minus sign will appear immediately to the left of the $ sign.

Example:

```
PRINT USING "**$##.##";2.34
 ***$2.34
```

, (comma) to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma at the end of the format string is printed as part of the string. A comma specifies another digit position. It has no effect if used with the exponential (^^^^) format.

Examples:

```
PRINT USING "####,.##";1234.5
 1,234.50

PRINT USING "####.##,";1234.5
 1234.50,
```

^^^^ (carets) may be placed after the digit position characters to specify exponential format. The four carets allow space for E±xx or D±xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

Examples:

```
PRINT USING "##.##^^^^";234.56
 2.35E+02

PRINT USING ".####^^^^-";-888888
 .8889E+06-
```

```
PRINT USING "+.##^^^^";123
 +.12E+03
```

_ (underscore) in the format string causes the next character to be output as a literal character.

Example:

```
PRINT USING "_!##.##_!";12.34
 !12.34!
```

The literal character itself may be an underscore by placing "__" (two underscores) in the format string.

% is printed in front of the number if the number to be printed is larger than the specified numeric field. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

Examples:

```
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 **v$**;]**list of expressions**

PURPOSE Writes data sequentially to a file.

■ ■ ■

filenum is the number used when the file was opened.

v$ consists of formatting characters as described in PRINT USING statement.

list of expressions consists of the numeric and/or string expressions that will be written to the file.

PRINT # does not compress data on the file. An image of the data is written to the file just as it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data on the file as noted below so that it will be input correctly from the file. (Note that using WRITE # avoids the difficulties referred to below since WRITE # inserts its own delimiters.)

In the list of expressions, numeric expressions should be delimited by semicolons. For example:

```
PRINT #1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks inserted between print fields will also be written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT#1,A$;B$
```

would write

```
CAMERA93604-1
```

to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

This is written to the file as

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, use CHR$(34) to write them to the file with explicit quotation marks.

For example, let A$="CAMERA, AUTOMATIC" and let B$=" 93604-1".

The statement

```
PRINT #1,A$;B$
```

would write the following to the file:

```
CAMERA, AUTOMATIC   93604-1
```

and the statement

```
INPUT #1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly in the file, use CHR$(34) to write double quotes to the file. The statement

```
PRINT #1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image:

```
"CAMERA, AUTOMATIC""   93604-1"
```

and the statement

```
INPUT #1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT # statement may also be used with the USING option to control the format of the file. For example:

```
PRINT #1,USING"$$###.##,";J;K;L
```

See also Appendix A, Sequential and Random Files.

PSET Statement

FORMAT PSET [STEP](**x-coordinate,y-coordinate**) [**,color**]

PURPOSE Draws a specified point on the screen (default color is the foreground color).

■ ■ ■

This statement can only be used in graphics mode (SCREEN 1, 2, 104 or 105).

x-coordinate and **y-coordinate** specify the point on the screen.

color is the number of the color to be used. In medium resolution (SCREEN 1), **color** ranges from 0 to 3, where 0 indicates the background color and 1 to 3 denote colors from the current palette (see Table 7-2). In high resolution (SCREEN 2) and super resolution (SCREEN 104 and 105), **color** can be either 0 (background color) or 1 (foreground color). Default is 3 for medium resolution and 1 for high- and super resolution. Default for monochrome screens is the foreground color.

The STEP option, if used, means that **x-coordinate** and **y-coordinate** are relative to the most recent cursor location. For example, if the most recent location was (10,10), then STEP (10,5) would reference the point at (20,15).

When GWBASIC scans coordinate values, it will allow them to be beyond the edge of the screen. (The size of the screen can be adjusted with the WIDTH statement.) However, values outside the integer range -32768 to 32767 will cause an "Overflow" error.

PSET allows the color to be omitted from the command line. If it is omitted, the default is the foreground color.

```
1 SCREEN 105
5 REM DRAW A LINE FROM (0,0) TO (100,100)
10 FOR I=0 TO 100
20 PSET (I,I)
30 NEXT
35 REM NOW ERASE THAT LINE
40 FOR I=0 TO 100
50 PSET STEP (-1,-1),0
60 NEXT
```

This example draws a line from (0,0) to (100,100) and then erases that line by overwriting it with the background color.

PUT Statement (Files)

FORMAT PUT **[#]filenum[,number]**

PURPOSE Writes a record from a random buffer to a random file.

■ ■ ■

filenum is the number under which the file was opened.

number is the number for the record to be written, in the range 1 to 16,777,215. If it is omitted, the record will be written in the next available record number (after the last PUT).

PRINT #, PRINT # USING, WRITE #, LSET, and RSET may be used to put characters in the random file buffer before executing a PUT statement. With WRITE #, GWBASIC adds spaces in the buffer up to the carriage return.

Reading or writing past the end of the buffer will cause a "Field overflow" error.

GWBASIC may buffer the data written to a file, and defer the disk access until it has buffered 512 characters.

For communications files, **number** is the number of bytes to write to the file, which cannot exceed the value set by the /S: switch on the GWBASIC command.

The GET and PUT statements allow fixed-length input and output for COM files. However, because of the low performance associated with telephone line communications, it is not advisable to use GET and PUT with COM files transmitted over telephone lines.

For an example, see GET Statement (Files).

PUT Statement (Graphics)

FORMAT PUT **(x,y),array[,action]**

PURPOSE Writes a graphic image from an array onto a specified area of the screen.

■ ■ ■

This form of the PUT statement is used in graphics mode only (SCREEN 1, 2, 104 and 105).

PUT transfers back to the screen a graphics image that was written to an array by the GET statement (graphics).

(x,y) specifies the point where a stored image is to be displayed on the screen. The specified point is the coordinate of the top left corner of the image. If the image to be transferred is too large to fit on the screen, an "Illegal function call" error will result.

array is the name of a numeric array that contains the information to be transferred. Specific information about array is given in GET statement (graphics).

action is PSET, PRESET, AND, OR, or XOR. The default is XOR.

PSET transfers data point by point onto the screen. Each point has the color attribute that it had when written to the array by GET.

PRESET is the same as PSET, except that PRESET causes the image to appear in its complementary colors. In super- and high-resolution modes, the black and white are simply reversed. For medium-resolution color graphics, where points have color attributes in the range 0 to 3, points are plotted as follows:

Attribute in array	Plotted with attribute
0	3
1	2
2	1
3	0

AND is used when the image is to be transferred over an existing image on the screen.

OR superimposes the image onto the existing image.

XOR is a special mode often used for animation. It causes the points on the screen to be inverted where a point exists in the array image. This behavior is exactly like that of the cursor. When an image is PUT against a complex background twice, the background is restored unchanged. This allows you to move an object around the screen without obliterating the background.

The default action is XOR.

Table 7-8 contains the effects of the PUT statement when used in medium-resolution graphics mode (SCREEN 1).

Table 7-8

EFFECTS OF AND, OR, AND XOR ON COLOR IN MEDIUM RESOLUTION

Action	Array Value	Screen Color			
		0	1	2	3
AND	0	0	0	0	0
	1	0	1	0	1
	2	0	0	2	2
	3	0	1	2	3
OR	0	0	1	2	3
	1	1	1	3	3
	2	2	3	2	3
	3	3	3	3	3
XOR	0	0	1	2	3
	1	1	0	3	2
	2	2	3	0	1
	3	3	2	1	0

Thus for example with XOR, an array element with a color attribute of 1 ("Array Value" column) will be PUT on the screen with a color as follows: if the screen color at that point was 0, the color will change to 1, color 1 will change to 0, 2 to 3 and 3 to 2.

Use the following steps to animate an object:

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. Go to step 1, but this time PUT the object(s) at the new location.

Movement done this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1 and by making sure that there is enough time delay between 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using PSET. The idea is to leave a border around the image that is as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points. This method may be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

The following example shows how you can use the PUT statement for animation. It uses PSET to move the figure across the screen. Substitute the other values of **action** to see what effect they have.

```
10 CLS: SCREEN 105
20 DIM A(200)
30 PI = 3.14159
40 CIRCLE (320,160),30,,-0.1*PI,-1.9*PI
50 CIRCLE (330,150),3
60 GET (280,120) - (360,190),A: CLS
70 X = 280: Y = 120
80 FOR N = 1 TO 20
90 PUT (X,Y),A,PSET
100 X = X + 8
110 NEXT N
```

RANDOMIZE Statement

FORMAT RANDOMIZE [**expression**]
RANDOMIZE TIMER

PURPOSE Reseeds the random number generator.

■ ■ ■

expression is the random number seed, with a value in the range -32768 to 32767.

If **expression** is omitted, GWBASIC suspends program execution and prompts

```
Random Number Seed (-32768 to 32767)?
```

before executing RANDOMIZE.

If **expression** is an integer or numeric variable, the value of the integer or variable is used to seed the random numbers.

The TIMER option allows you to use the TIMER function to pass a random number seed.

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 every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the seed with each run.

```
10 RANDOMIZE
20 INPUT "NUMBER OF RANDOM NUMBERS ";N
30 FOR I = 1 TO N
40 PRINT RND;
50 NEXT I
Ok
RUN
Random number seed (-32768 to 32767)? 6
NUMBER OF RANDOM NUMBERS ? 3
 .4417627   .1085309   .182628
Ok
```

READ Statement

FORMAT READ **variable[,variable]...**

PURPOSE Reads values from a DATA statement and assigns them to variables. (See DATA statement.)

■ ■ ■

variable is a numeric or string variable or array element that will receive the value read.

A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in the list exceeds the number of elements in the DATA statement(s), an "Out of data" error message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread data from a DATA statement, use RESTORE (see RESTORE statement).

Example 1

```
    .
    .
    .
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 segment reads the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2

```
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "PASADENA,", CALIFORNIA, 91127
40 PRINT C$,S$,Z
Ok
RUN
 CITY           STATE          ZIP
 PASADENA,      CALIFORNIA     91127
Ok
```

This program reads string and numeric data from the DATA statement in line 30. Note that PASADENA requires quotes because of the comma following it, whereas no quotation marks around CALIFORNIA are necessary.

REM Statement

FORMAT REM **remark**

PURPOSE Allows explanatory remarks to be inserted in a program.

■ ■ ■

remark may be any sequence of characters.

REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into from a GOTO or GOSUB statement. Execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM. Do not use the single quotation form of REM in a DATA statement, as it would be considered valid data.

NOTE: When using remarks in GWBASIC, they stay resident in memory, reducing the memory available for program and data.

```
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
```

The above are examples of the use of REM.

RENUM Command

FORMAT RENUM **[newnum][,[oldnum][,increment]]**

PURPOSE Renumbers program lines.

■ ■ ■

newnum is the first line number to be used in the new sequence. The default is 10.

oldnum is the line in the current program where renumbering is to begin. The default is the first line of the program.

increment is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ELSE, ON...GOTO, ON...GOSUB, RESTORE, RESUME, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number yyyyy in xxxxx" is printed. The incorrect line number reference yyyyy is not changed by RENUM, but line number xxxxx may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

RENUM

The preceding command will renumber the entire program. The first line number will be 10, and succeeding lines will be numbered in increments of 10.

RENUM 300,,50

The entire program will be renumbered, the first line number will be 300, and succeeding lines will be numbered in increments of 50.

```
RENUM 1000,900,20
```

The program will be renumbered from line 900, the first new line number will be 1000, and each succeeding line will be incremented by 20.

RESET Command

FORMAT RESET

PURPOSE Closes all files on all drives.

■ ■ ■

This command closes all open files on all drives and writes the directory track to every disk with open files.

All files must be closed before a disk is removed from its drive.

```
998 RESET
999 END
```

Closes all open files before ending the program.

RESTORE Statement

FORMAT RESTORE **[line number]**

PURPOSE Allows DATA statements to be reread from a specified line.

■ ■ ■

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If **line number** is specified, the next READ statement accesses the first item in the specified DATA statement.

See GOTO statement.

```
10 DATA 1,2,3
20 DATA 4,5,6
30 READ A,B,C
40 PRINT A,B,C
50 READ A,B,C
60 PRINT A,B,C
70 RESTORE 20
80 READ A,B,C
90 PRINT A,B,C
Ok
RUN
 1             2             3
 4             5             6
 4             5             6
Ok
```

RESUME Statement

FORMAT
```
RESUME
RESUME 0
RESUME NEXT
RESUME line number
```

PURPOSE Continues program execution after an error recovery procedure has been performed.

■ ■ ■

Use one of the four formats shown above according to where execution is to resume.

RESUME or RESUME 0 causes execution to resume at the statement that caused the error.

RESUME NEXT causes execution to resume at the statement immediately following the one causing the error.

RESUME **line number** causes execution to resume at the **line number** specified.

If RESUME statement is not in an error-handling subroutine, a "RESUME without error" message is printed.

See GOTO statement.

```
10 ON ERROR GOTO 900
    .
    .
    .
900 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY
    AGAIN": RESUME 80
```

In the above, when error 230 occurs in line 90, RESUME causes the program to return to line 80.

RETURN Statement

FORMAT RETURN **[line number]**

PURPOSE Causes GWBASIC to return to the statement following the most recent GOSUB statement.

■ ■ ■

If **line number** is specified, execution will return to the specified line number in the program. This option should be used with extreme caution.

RIGHT$ Function

FORMAT **v$ = RIGHT$(a$,x)**

PURPOSE Returns the rightmost **x** characters of string **a$**.

■ ■ ■

a$ is any string expression.

x is an integer that specifies how many characters will be in the result.

If **x** is greater than or equal to the number of characters in **a$**, then RIGHT$ returns **a$**. If **x=0**, the null string (length zero) is returned.

See also the MID$ and LEFT$ functions.

```
10 TEST$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
20 FOR I=1 TO 5
30 PRINT RIGHT$(TEST$,I)
40 NEXT I
RUN
 Z
 YZ
 XYZ
 WXYZ
 VWXYZ
Ok
```

The RUN command causes five lines to be printed, each returning an additional character from the right end of the string.

RMDIR Command

FORMAT RMDIR **pathname**

PURPOSE Removes (deletes) an existing directory.

■ ■ ■

pathname specifies the name of the directory to be removed, and is a string of up to 63 characters, which must be enclosed in quotes. RMDIR works exactly like the DOS command RMDIR - see the DOS manual.

The directory to be removed must be empty of any files except the working directory (".") and the parent directory (".."), otherwise a "Path not found" or "Path/file access error" message will be displayed.

```
RMDIR "\SALES"
```

This statement causes the SALES directory on the current drive to be deleted.

RND Function

FORMAT `v = RND[(x)]`

PURPOSE Returns a random number between 0 and 1.

■ ■ ■

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded. (See RANDOMIZE statement.)

If `x` < 0, RND restarts the same sequence for any given `x`.

If `x` > 0 or if `x` is omitted, RND generates the next random number in the sequence.

If `x` = 0, the last number generated is repeated.

```
10 RANDOMIZE 531
20 PRINT RND*100
RUN
 79.51105
Ok
```

The above shows how the RND call can produce a value between 0 and 100.

RUN Command

FORMAT RUN **[line]**

RUN **filespec[,R]**

PURPOSE Executes the program currently in memory, or loads a program into memory and runs it.

■ ■ ■

line is the number of the line in the program where execution is to begin.

filespec is a file specification (see Section 3, under "File and Device Information") in the form of a string constant, and identifies the file containing the program to be run. The filespec consists of a filename or pathname with an optional device name (which must be that of a diskette or hard drive). The filename is the name used when the file was saved (remember that GWBASIC supplies the extension .BAS if none is given). If the device name is omitted, the DOS default drive is used.

If **line** is omitted, program execution begins at the lowest line number.

RUN **filespec** loads a file from disk into memory and runs it. Before loading the designated program, RUN closes all open files and deletes the current contents of memory.

With the "R" option, all data files remain open.

GWBASIC always returns to command level after a RUN command.

```
RUN "B:NEWFIL",R
```

The program NEWFIL is loaded from drive B: and run, with the files kept open.

SAVE Command

FORMAT SAVE **filespec[,A]**

SAVE **filespec[,P]**

PURPOSE Saves a program file on disk.

■ ■ ■

filespec is a file specification (see Section 3, under "File and Device Information") in the form of a string constant, and identifies the file containing the program to be saved. The filespec consists of a filename or pathname with an optional device name (which must be that of a diskette or hard drive). If the file name is eight characters or less and no extension is supplied, the extension .BAS is added to the name. If the device name is omitted, the DOS default drive is used.

If **filespec** already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, GWBASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect 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.

See also Appendix A, Sequential and Random Files.

```
SAVE "A:COM2",A
```

This example saves COM2.BAS on drive A: in ASCII.

```
SAVE "B:PROG",P
```

The above saves PROG on drive B: and protects it so it may not be altered.

SCREEN Function

FORMAT **v = SCREEN(row,col[,z])**

PURPOSE Returns the color or the ASCII code of the character at the specified row (line) and column of the active screen.

■ ■ ■

row is a number in the range 1 to 25.

col is a number in the range 1 to 40 or 1 to 80 depending on WIDTH.

z is a numeric expression that evaluates to a true or false value. It is valid only in text mode. If **z** evaluates to 0 (false), the ASCII code for the character is returned. Appendix C lists the possible codes.

If **z** is included and is non-zero, the color attribute is returned instead of the ASCII code. This attribute will be a number **n** in the range 0 to 255, and is interpreted in the following way:

(**n** MOD 16) = **f**, where **f** is the attribute of the foreground color

(((**n** - **f**)/16) MOD 128) is the attribute of the background color

The expression (**n** > 127) returns the value -1 (true) if the character is blinking, or 0 (false) if it is steady.

See Table 7-1 under the COLOR Statement (Text) to interpret the color attributes.

See also SCREEN statement.

```
10 A = SCREEN (20,21)
```

If the character at point 20,21 is X, then A is 88.

```
110 A = SCREEN (10,10,1)
```

A will be the color attribute of the character at location (10,10).

SCREEN Statement

FORMAT SCREEN [**mode**][,[**burst**][,[**apage**][,**vpage**]]]

PURPOSE Sets screen attributes for use in subsequent statements.

■ ■ ■

mode may be any of the following:

0 text mode at current width (40 or 80). It displays on the "current" monitor, as previously selected. This is the default value. (Note that in text mode WIDTH 40 only applies to monitors with color/graphics adapters present.)

1 medium-resolution graphics mode (320x200). For 325-line systems, the display is on the upper left side of the monochrome screen unless there is a color/graphics monitor adapter. Default screen width for this mode is 40 characters.

2 high-resolution graphics mode (640x200). For 325-line systems, the display is on the upper two-thirds of the monochrome screen unless there is a color/graphics monitor adapter. (Note that in high-resolution graphics only two colors, black and white, are possible, even on monitors with color/graphics adapters present.)

NOTES: For 400-line systems, medium- and high-resolution graphics modes are emulated to cover the entire screen. If a color/graphics monitor adapter is present, emulation ceases and the modes automatically switch the display to a color monitor.

Programs written for medium-resolution color graphics emulation may not be displayed as expected, because the monochrome screen has only two colors (black and white).

100 forces text mode onto the monochrome screen (that is, there is a switch from color to monochrome). If the screen is already monochrome, the statement is ignored.

101 switches from monochrome display to color/graphics adapter. On 325-line systems only, if no color/graphics monitor adapter is available, an "Illegal function call" error message is displayed.

104 sets monochrome screen for super-resolution graphics (640 x 400 or 640 x 325 lines) and text. In this mode, text written to the screen over an existing graphics image will overwrite that part of the image (to avoid this, write the text to the screen first). Text characters in this mode are actually graphics characters and have a different appearance from text in other modes.

105 sets monochrome screen for super-resolution graphics (640 x 400 or 640 x 325 lines) and text. In this mode, text written to the screen over an existing graphics image will merely be superimposed on that image.

NOTE: Modes 104 and 105 produce true super-resolution graphics, and are not emulation modes.

burst is either set to zero, which disables color, or to any non-zero value, which enables color. Note that **burst** defaults to non-zero with an RGB monitor. **mode** and **burst** have the following effect on color when a color/graphics monitor adapter is present:

Mode	Burst	Effect on Color
0	0	Disabled
	1	Enabled
1	0	Enabled
	1	Disabled
2	No effect	-
100	No effect	-
101	0	None
104	No effect	-
105	No effect	-

apage (active page) specifies the current active graphics page, the one being written to memory, and is a number between 0 and 15 - see Section 4 for the formula to calculate the maximum number of graphic pages. With a color/graphics adapter, however, **apage** refers to a text page and is a number between 0 to 7 for width 40 or between 0 to 3 for width 80. It is only used with mode 0.

vpage (visual page) specifies the current visual page, the one being written on the screen, and is a number between 0 and 15, as for **apage**. With a color/graphics adapter, **vpage** is valid only in text mode, and if omitted, defaults to **apage**.

Note that while **vpage** and **apage** will often be the same, you may display one page on the screen while writing to another, and therefore **vpage** may be different from **apage**.

If **mode** is set to 0, 104, or 105 with only **apage** and **vpage** specified, display pages can be changed for screen viewing. By manipulating these parameters you can display one page while building another and then switch visual pages instantly.

The default graphics page (for SCREEN 104 and 105) is determined by GWBASIC at load time - see Section 4 for more details. With a color/graphics adapter, the default text page is 0.

If the new screen mode is the same as the previous mode, only the new parameters are updated.

The memory address for any given super-resolution graphics page is [page*800H]:0000. You will need to input this address in a DEF SEG statement prior to using a BLOAD statement.

If you wish to interchange active pages, the cursor position on the current page should be stored using POS(0) and CSRLIN, because the cursor is shared among all pages. Cursor position can then be restored with LOCATE.

Any parameter may be omitted. Except for **vpage**, omitted parameters assume the old value.

For both the standard monochrome display and the color/graphics monitor adapter it is suggested that SCREEN 0,0,0 and WIDTH 80 statements be used at the beginning of a program.

Out-of-range values cause an "Illegal function call" error, and previous values are retained.

```
10 SCREEN 105,,5,5
20 CLS
```

Line 10 enables both text and super-resolution graphics to be displayed and written to page 5. Line 20 clears both alphanumeric and graphic screens in memory.

```
10 SCREEN 0,1,0,0
```

The above is for a color/graphics monitor adapter. It selects text mode with color and sets the active and visual page to 0.

SGN Function

FORMAT `v = SGN(x)`

PURPOSE Returns the sign of `x`.

■ ■ ■

`x` is any numeric expression.

If `x` > 0, SGN(`x`) returns 1.
If `x` = 0, SGN(`x`) returns 0.
If `x` < 0, SGN(`x`) returns -1.

```
ON SGN(X)+2 GOTO 100,200,300
```

The program 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 Used to exit the BASIC program, execute a DOS command or run a program such as a .EXE or .BAT program, and return to the BASIC program at the line after the SHELL statement.

■ ■ ■

command-string is a string expression containing the name of the command or program to be executed, optionally followed by parameters. Any text separated from the name by at least one blank is regarded as parameters.

If you omit **command-string**, the DOS prompt is displayed and you can execute one or more COMMAND.COM commands before typing EXIT to return to the BASIC program.

A program name in **command-string** may have any extension you wish. If no extension is supplied, the system looks for a .COM file, then a .EXE file, and finally, a .BAT file. If none is found, SHELL will issue a "File not found" error.

A program or command executed in this way under the SHELL statement is called a "child process". Child processes are executed by SHELL loading and running a copy of COMMAND.COM with the "/C" switch. By using COMMAND.COM in this way, any parameters you may have are passed to the default file control blocks. Standard input and output may be redirected, and built-in commands such as DIR, PATH, and SORT may be executed.

BASIC remains in memory while the child process is running. When the child finishes, BASIC continues.

You cannot use SHELL to call BASIC. If you try to do this the system will issue the message: "You cannot Shell to BASIC" and return you to the parent BASIC.

When you execute a child process using SHELL, you have to be careful to ensure that the child process does not change anything that the BASIC program was using, such as closing files that were open when SHELL was executed. The following set of guidelines will help you to avoid potential disasters.

1. Hardware

   The child process may change the screen mode. The simplest way to guard against this is to use a SCREEN statement followed by CLS directly after the SHELL statement in the BASIC program, ensuring that SCREEN specifies the correct screen mode for the remainder of the BASIC program.

   Save and restore any interrupt vectors that the child process uses. The child process itself can perform this task.

   Some of the internal devices on the main PCB are set to a specific state by GWBASIC, and must not be changed by the child process. These are the 8259 interrupt controller, the 8253 interval timer, the 8237 DMA controller, the 8255 peripheral interface and the 8250 UART. For further information about these devices, see the Technical Reference manual for your system.

2. The File System

   A child process which alters any file that was OPENed by the parent BASIC program may cause unpredictable effects. If you need to update such files, close them in the BASIC program before using SHELL, then reopen them on returning to the BASIC program.

3. Memory Management

   Before BASIC executes a SHELL statement, it will try to free any memory it is not then using, unless the original BASIC command was given with the /M: switch. If the /M: switch is set, BASIC assumes that you intended to load something just above BASIC's work space. This prevents BASIC from compressing its work space before doing the SHELL. For this reason SHELL may fail with an "Out of memory" error when using the /M: switch.

   A better method is to load assembly language subroutines before BASIC is run. You can achieve this by placing code at the end of the subroutines that uses interrupt 27H to allow them to exit to DOS and stay resident. For example:

```
CSEG      SEGMENT CODE
          .
          .           ;(assembly language
                        subroutine)
          .
RET                   ;last instruction
START::
INT       27H         ;terminate, stay resident
CSEG      ENDS
END       START
```

Be sure to "load" these subroutines by running them before invoking BASIC. The AUTOEXEC.BAT file is very useful for this purpose.

A child process should never terminate and stay resident. Doing so may not leave BASIC enough room to restore its work space to the original size. If BASIC cannot restore the work space, all files are closed, the error message "SHELL can't continue" is displayed, and BASIC exits to DOS.

The following example shows how you can exit to DOS from a BASIC program, use some DOS commands and then resume the program.

```
    .
(BASIC program)
    .
  SHELL     (exits to DOS, displaying the A> prompt)

A>DIR       (user types DIR to see files)

A>EXIT      (user types EXIT to return to BASIC)
```

The next example creates a file to contain data to be sorted, exits to DOS to run the SORT utility and returns to BASIC.

```
10 OPEN "SORTIN.DAT" FOR OUTPUT AS #1
                    .
   (program writes data to be sorted into SORTIN.DAT)
                    .
1000 CLOSE 1
1010 SHELL "SORT <SORTIN.DAT >SORTOUT.DAT"
1020 OPEN "SORTOUT.DAT" FOR INPUT AS #1
                    .
   (program processes the sorted data)
                    .
```

SIN Function

FORMAT `v = SIN(x)`

PURPOSE Returns the sine of `x`.

■ ■ ■

`x` must be in radians.

SIN(`x`) is calculated in single precision. See also COS function.

```
PRINT SIN(1.5)
 .9974951
Ok
```

SOUND Statement

FORMAT SOUND **freq,duration**

PURPOSE Generates sound through the speaker.

■ ■ ■

freq is the frequency in Hertz (cycles per second) and in the range 37 to 32767.

duration is a numeric expression in the range 0 to 65535 indicating the desired length in clock ticks, which occur 18.2 times per second, or 1092 times per minute.

See Table 7-9 for information on the frequencies generated by this statement. Refer to the PLAY statement for information on producing continuous notes with no pauses between statements.

Table 7-9

NOTE FREQUENCIES FOR FOUR OCTAVES

Note	Frequency	Note	Frequency
C	130.8	C	523.3
D	146.8	D	587.3
E	164.8	E	659.3
F	174.6	F	698.5
G	196.0	G	784.0
A	220.0	A	880.0
B	246.9	B	987.8
C*	261.6	C	1046.5
D	293.7	D	1174.7
E	329.6	E	1318.5
F	349.2	F	1396.9
G	392.0	G	1568.0
A	440.0	A	1760.0
B	493.9	B	1975.5
*Middle C			

Doubling the frequency of a note gives the same note an octave higher; halving the frequency gives the same note an octave lower.

If a rest is desired, use 32767, **duration**.

If **duration** is zero, any SOUND statement that is currently executing will be terminated.

Table 7-10 lists some typical tempos.

Table 7-10

TEMPO CALCULATIONS

Beats/ Minute	Tempo	Ticks/ Beat
	Larghissimo (very slow)	
40-60	Largo	27-18
60-66	Larghetto	18-17
	Grave	
	Lento	
66-76	Adagio	17-14
	Adagietto (slow)	
76-108	Andante	14-10
	Andantino (medium)	
108-120	Moderato	10-9
	Allegretto (fast)	
120-168	Allegro	9-7
	Vivace	
	Veloce	
168-208	Presto	7-5
	Prestissimo (very fast)	

The following example creates an endless loop of random sounds.

```
30 SOUND RND*1000 + 37, 2
40 GOTO 30
```

SPACE$ Function

FORMAT **v$** = SPACE$(**x**)

PURPOSE Returns a string of **x** spaces.

■ ■ ■

x must be an integer in the range 0 to 255.

See also SPC function.

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
 1
  2
   3
    4
     5
Ok
```

Each number I follows I spaces on a line. GWBASIC puts a space in front of positive numbers, which results in I+1 leading spaces.

SPC Function

FORMAT PRINT SPC(**x**)

PURPOSE Skips **x** spaces in a PRINT statement.

■ ■ ■

n must be in the range 0 to 255.

SPC may only be used with PRINT, LPRINT, and PRINT # statements.

GWBASIC acts as if SPC has an implied semicolon after it. Therefore, if SPC occurs at the end of a list of data items, GWBASIC does not add a carriage return.

See also SPACE$ function.

```
PRINT "LEFT" SPC(15) "RIGHT"
 LEFT                RIGHT
Ok
```

There are 15 spaces between LEFT and RIGHT.

SQR Function

FORMAT v = SQR(x)

PURPOSE Returns the square root of x.

x must be greater than or equal to zero.

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.162278
 15            3.872984
 20            4.472136
 25            5
Ok
```

The above is a way to calculate the square roots of 10, 15, 20, and 25.

STICK Function

FORMAT **v** = STICK(**n**)

PURPOSE Accepts input from the joystick in the form of x- and y-coordinates.

■ ■ ■

n is an integer from 0 to 3 as follows:

0 = the x-coordinate of joystick A

1 = the y-coordinate of joystick A

2 = the x-coordinate of joystick B

3 = the y-coordinate of joystick B

All four parameters will be assigned by first calling STICK(0). The calls to functions STICK(1) through STICK(3) will then return the values collected by the STICK(0) function. This is required to allow all input parameters to reflect the position of the STICK at a particular instant.

```
10 FOR I = 1 TO 3
20 FOR J = 0 TO 3
30 PRINT STICK(J);
40 NEXT J
50 PRINT
60 NEXT I
Ok
```

STOP Statement

FORMAT STOP

PURPOSE Terminates program execution and returns to command level.

■ ■ ■

STOP statements may be used anywhere in a program. A STOP causes the following message to be printed:

```
Break in line nnnnn
```

Unlike the END statement, the STOP statement does not close files.

GWBASIC returns to command level after a STOP is executed. Resume program execution with CONT (see CONT command).

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.76923
Ok
CONT
 115.9
Ok
```

The above shows how STOP can be used to allow you to examine variables in the GWBASIC program.

STR$ Function

FORMAT `v$ = STR$(x)`

PURPOSE Returns a string representation of the value of **x**.

■ ■ ■

x is any numeric expression.

If **x** is positive, the string returned will contain a leading blank, which is the space reserved for the minus sign when the number is negative.

See also the VAL function.

```
5 REM arithmetic for kids
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
    .
    .
    .
```

The program branches to different subroutines depending on the number entered. STR$ converts the number to a string, and branching is based on the length of the string.

STRIG Statement and Function

FORMAT
```
STRIG ON
STRIG OFF
STRIG STOP
v = STRIG (n)
```

PURPOSE Returns the status of the specified joystick trigger or enables, disables, or stops trapping of the joystick.

■ ■ ■

v is a numeric variable for storing the result of the function.

n is a number from 0 to 3, designating which trigger is to be checked, as follows:

0 Returns -1 if trigger A was pressed since the last STRIG(0) statement; returns 0 if not.

1 Returns -1 if trigger A is currently down; returns 0 if not.

2 Returns -1 if trigger B was pressed since the last STRIG(2) statement; returns 0 if not.

3 Returns -1 if trigger B is currently down; returns 0 if not.

STRIG ON enables trapping of joystick activity.

STRIG STOP disables trapping, but if the joystick trigger is pressed, that event will be remembered and trapped as soon as there is a STRIG ON.

STRIG OFF not only disables trapping, it also does not remember a subsequent event (i.e., if the trigger is pressed).

STRIG ON requires an ON STRIG statement (see ON STRIG statement). While trapping is enabled, and if a nonzero line number is specified in the ON STRIG statement, GWBASIC checks between every statement to see if the joystick trigger has been pressed.

When a trap occurs, that occurrence of the event is destroyed. Therefore the x=STRIG(n) function will always return false inside a subroutine, unless the event has been repeated since the trap. So if you wish to perform different procedures for various joysticks, you must set up a different subroutine for each joystick, rather than including all the procedures in a single subroutine.

```
10 IF STRIG(0) THEN BEEP
20 GOTO 10
```

An endless loop is created to beep whenever the trigger button on joystick 0 is pressed.

STRING$ Function

FORMAT **v$** = STRING$(**n**,**m**)

v$** = STRING$(n**,**a$**)

PURPOSE Returns a string of length **n** whose characters all have ASCII code **m** or the first character of **a$**.

■ ■ ■

n,**m** are in the range 0 to 255.

a$ is any string expression.

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
 ----------MONTHLY REPORT----------
```

Before and after the character string "MONTHLY REPORT", this program prints 10 hyphens (as stored in string variable X$).

SWAP Statement

FORMAT SWAP **variable1,variable2**

PURPOSE Exchanges the values of two variables.

■ ■ ■

variable1,variable2 are the names of two variables or array elements.

Any type of variable may be swapped (integer, single-precision, double-precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

If the second variable is not already defined when SWAP is executed, an "Illegal function call" error will result.

```
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
 ONE FOR ALL
 ALL FOR ONE
Ok
```

Line 30 causes A$ to have the value ALL and B$ to have the value ONE.

SYSTEM Command

FORMAT SYSTEM

PURPOSE Returns control to DOS.

■ ■ ■

This command performs a "warm" boot, i.e., all open files are closed, and control is returned to DOS.

TAB Function

FORMAT [PRINT] TAB(x)

PURPOSE Moves cursor or print head to position x.

■ ■ ■

x must be in the range 1 to 255.

If the current print position is already beyond space x, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the defined width.

TAB may only be used in PRINT, LPRINT, and PRINT # statements.

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
 NAME                      AMOUNT

 G. T. JONES               $25.00
Ok
```

TAB causes the NAME and AMOUNT columns to line up.

TAN Function

FORMAT v = TAN(x)

PURPOSE Returns the tangent of x.

■ ■ ■

x is an angle, which must be given in radians. To convert from degrees to radians, multiply by PI/180, where PI = 3.141593.

TAN(x) is calculated in single precision.

If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

```
10 Y=Q*TAN(X)/2
```

Y is equal to Q times the tangent of X divided by 2.

TIME$ Statement

FORMAT TIME$ = **x$**

PURPOSE Sets the time.

■ ■ ■

x$ returns one of the following strings:

hh sets the hour (in the range 0 to 23); minutes and seconds default to 00.

hh:mm sets the hour and minute; seconds default to 00.

hh:mm:ss sets the hour, minute, and second.

This statement complements the TIME$ variable, which retrieves the time.

```
10 TIME$="17:00:00"
```

The current time is set at 5:00 p.m.

TIME$ Variable

FORMAT **v$** = TIME$

PURPOSE Retrieves the current time.

■ ■ ■

This variable returns an eight-character string in the form **hh:mm:ss**, where **hh** is the hour (00 through 23), **mm** is the minute (00 through 59), and **ss** is the second (00 through 59). A 24-hour clock is used. Therefore 8:00 p.m. would be shown as 20:00:00.

To set the time from a BASIC program, use TIME$ statement. Note that the TIME$ variable also returns the time if set by the DOS command TIME.

```
10 PRINT TIME$
```

Prints the time set with the TIME$ statement.

TIMER Function

FORMAT v = TIMER

PURPOSE Returns the number of seconds elapsed since midnight or the last system reset.

■ ■ ■

TIMER is a read-only function returning a single-precision floating-point number which includes fractional seconds as precisely as possible.

If the system clock has been set, either via the TIME$ statement or via the DOS command TIME, the TIMER function returns the number of seconds since midnight, otherwise it returns the number of seconds since the last system reset.

TIMER Statement

FORMAT
TIMER ON
TIMER OFF
TIMER STOP

PURPOSE Enables (ON), disables (OFF) or suspends (STOP) timer event trapping (i.e., testing whether the timer has reached the value specified in an ON TIMER(**n**) statement).

■ ■ ■

ON TIMER(**n**) must first be set with a line number and TIMER ON must be executed before GWBASIC checks whether a timer event has occurred. If so, the program will perform a GOSUB to the trap routine at the line number specified in ON TIMER(**n**).

TIMER STOP suspends trapping, but if a subsequent TIMER ON statement is executed the GOSUB is performed when the event next occurs.

TIMER OFF stops trapping altogether. If a subsequent TIMER ON statement is executed the GOSUB is not performed.

TRON and TROFF Commands

FORMAT TRON

TROFF

PURPOSE Traces the execution of program statements.

■ ■ ■

As an aid in debugging, the TRON command enables a trace flag that displays each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF or NEW commands.

```
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
```

TRON and TROFF are used to trace execution of a loop. Line numbers are in brackets. The other numbers are the values of J, K, and L as printed by the program in line 40.

USR Function

FORMAT **v = USR[n](arg)**

PURPOSE Calls an assembly language subroutine.

■ ■ ■

n is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. (See DEF USR statement.) If **n** is omitted, USR0 is assumed.

arg is any numeric expression or string variable that will be the argument to the assembly language 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.

If a segment other than the default 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.

NOTE: While the USR function is provided for compatibility with existing programs, it is recommended that new programs that call assembly language subroutines should do so by means of the CALL or CALLS statements, which can pass multiple arguments. In addition, the CALL statement is compatible with more languages than the USR function.

```
100 DEF SEG = &H8000
110 DEF USR0 = 0
120 X = 5
130 Y = USR0(X)
140 PRINT Y
```

An assembly language routine at segment 8000 hex, offset 0, is called with the argument set to 5.

VAL Function

FORMAT v = VAL(x$)

PURPOSE Returns the numerical value of string x$.

■ ■ ■

x$ is a string expression.

The VAL function also strips leading blanks, tabs, and line feeds from the argument string. For example,

```
VAL(" -3")
```

returns -3.

VAL(x$) returns 0 if the first character of x$ is not numeric.

For numeric to string conversion, see STR$ function.

```
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"
```

In this example VAL is used to identify names of individuals living out of California (line 20). In line 30 VAL is used to identify names of individuals living in Long Beach, California.

VARPTR Function

FORMAT 1 **v** = VARPTR(**variable**)

FORMAT 2 **v** = VARPTR(**#filenum**)

PURPOSE Returns the starting address in memory of the variable or BASIC file control block.

■ ■ ■

Format 1

variable is a numeric, string, or array variable in the program. **variable** must have previously been assigned a value, or an "Illegal function call" error results.

VARPTR returns the address of the first byte in the data portion of the variable identified by **variable**. For string variables, the address of the first byte of the string descriptor is returned. Details of how variables are stored are given in Appendix G, Technical Hints.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly 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.

NOTE: All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

For both formats the address will be an integer in the range -32768 to 32767. If a negative address is returned, add it to 65536 to obtain the actual address.

Format 2

filenum is the number under which the file was opened.

For sequential files, the starting address of the disk I/O buffer assigned to **filenum** is returned. For random files, the address of the field buffer assigned to **filenum** is returned.

```
100 A=USR(VARPTR(B))
```

The above returns the offset address of the variable "B".

VARPTR$ Function

FORMAT **v$** = VARPTR$(**variable**)

PURPOSE Returns a string that defines the type of variable and its address in memory.

■ ■ ■

variable is the name of a numeric, string, or array variable existing in the program.

This function is primarily used to execute substrings with PLAY and DRAW in programs that will later be compiled. For programs that will not later be compiled, the standard syntax of the PLAY and DRAW statements is sufficient to produce the desired results.

A value must be assigned to **variable** before attempting this function, or an "Illegal function call" error results.

VARPTR$ returns a three-byte string in the form:

Byte 0 **type**

Byte 1 low byte of variable address

Byte 2 high byte of variable address

type indicates the variable type as follows:

2 integer
3 string
4 single-precision
8 double-precision

The returned value will be the same as CHR$ (**type**)+MKI$(VARPTR (**variable**)).

Since addresses of arrays change whenever a new simple variable is assigned, all simple variables should be assigned before calling VARPTR$ for an array element.

```
100 A$ = "ABCDEFG"
200 PLAY "x" + VARPTR$(A$)
```

In this example, VARPTR$ has been used to indicate the string "ABCDEFG". Thus the subcommand X (execute) plus the contents of A$ are used as the string expression in the PLAY statement.

VIEW Statement

FORMAT VIEW [[SCREEN][(x1,y1)-(x2,y2)[,[color][,[border]]]]]

PURPOSE Defines a screen viewport (a subset of the screen area) for subsequent graphics display.

■ ■ ■

This statement can only be used in graphics mode (SCREEN 1, 2, 104 or 105).

x1,y1 - x2,y2 are the upper left and lower right coordinates respectively of the section of the screen that is to be used as the viewport. The coordinates must be within the physical screen area.

color allows you to fill the viewport with the specified color. In medium resolution (SCREEN 1), **color** ranges from 0 to 3, where 0 indicates the background color and 1 to 3 denote colors from the current palette (see Table 7-2). In high resolution (SCREEN 2) and super resolution (SCREEN 104 and 105), **color** can be either 0 (background color) or 1 (foreground color). If you omit this parameter, the viewport is not filled.

border is an integer expression identifying a color as above, and draws a border in the specified color around the viewport if space for a border is available. If you omit this parameter, no border is drawn.

The SCREEN option causes coordinates in subsequent graphics statements to be regarded as absolute to the physical screen area. If you omit the SCREEN option, such coordinates are taken as being relative to the specified viewport.

VIEW with no parameters defines the entire screen as the viewport.

If:

```
VIEW (10,10)-(200,100)
```

were executed, then the point plotted by the statement PSET (0,0),3 would be at the physical screen location 10,10 since the PSET coordinates are interpreted relative to the viewport.

If:

```
VIEW SCREEN (10,10)-(200,100)
```

were executed, then the point plotted by the statement PSET (0,0),3 would actually not appear because the PSET coordinates are interpreted as absolute to the physical screen area and 0,0 is outside the viewport. PSET (10,10),3 is within the viewport, and places the point in its upper-left hand corner.

A number of VIEW statements may be executed. If the newly-defined viewport is not wholly within the previous one, the screen can be re-initialized with the VIEW statement and the new viewport defined. If the new viewport is entirely within the previous one, as in the following example, the intermediate VIEW statement is not necessary. This example opens three viewports, each smaller than the previous one. In each case, a line that is defined to go beyond the borders is programmed, but only appears within the viewport boundaries.

```
260  CLS
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-line** TO **bottom-line]**

PURPOSE Defines a screen viewport (a subset of the screen area) for text display.

■ ■ ■

VIEW PRINT without top and bottom line parameters defines the whole screen area as the text window.

Use VIEW PRINT to define a text display area to coincide with a graphics viewport defined by the VIEW statement. Statements and functions which operate within the defined text area include CLS, LOCATE, PRINT and SCREEN. Note that changing screen mode (e.g., SCREEN 1, SCREEN 104) cancels the effect of VIEW PRINT.

The screen editor will limit functions such as scroll and cursor movement to the display area.

top-line can be equal to, but not more than, **bottom-line**.

```
10 CLS:VIEW PRINT 10 TO 15
20 FOR A = 1 TO 500
30 PRINT A;
40 NEXT
```

This example displays the numbers 1 to 500, scrolling them through a six-line text window in the center of the screen.

WAIT Statement

FORMAT WAIT **port,i[,j]**

PURPOSE Suspends program execution while monitoring the status of a machine input port.

■ ■ ■

port is the port number, an address from 0 to 1023 (&H3FF). Table 7-6 gives a list of port addresses.

i,j are integer expressions in the range 0 to 255.

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern.

The data read at the port is XORed with **j** and then ANDed with **i**. If the result is zero, GWBASIC loops back and reads the data at the port again. If the result is non-zero, execution continues with the next statement. If **j** is omitted, it is assumed to be zero.

In other words, WAIT allows you to test for either a 1 or a 0 at one or more bit positions at the specified input port. The positions tested are those having a 1 as indicated by the binary value of **i**. If you omit **j**, the indicated positions are tested for 1s. If you include **j**, a 1 in any bit position in **j** for which there is a 1 at the same position in **i** will cause WAIT to test for a 0 at that position.

Program execution is suspended while WAIT tests the specified bits. If any single one of the specified bits becomes 1 (or 0 if **j** is used), the program resumes execution.

CAUTION

It is possible to enter an infinite loop with WAIT, in which case it will be necessary to reboot with Ctrl-Alt-Del or a system reset. To avoid this situation, WAIT must have the specified value at **port** during some point in the program execution.

```
10 WAIT 40,8
```

WAIT suspends program execution until there is a 1 in the fourth bit position (i.e., 1000, the binary value of 8) in port 40.

WHILE and WEND Statements

FORMAT WHILE **expression**

.
.
.

(loop statements)

.
.
.

WEND

PURPOSE Executes a series of statements in a loop as long as a given condition is true.

■ ■ ■

expression is any numeric expression.

If expression is not zero (i.e., true), loop statements are executed until the WEND statement is encountered. GWBASIC then returns to the WHILE statement and checks **expression**. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE...WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

NOTE: Be careful not to direct program flow into a WHILE/WEND loop without entering through the WHILE statement.

```
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
```

There are J elements in character string A$, and this program sorts them into alphabetical order.

WIDTH Statement

FORMAT 1 WIDTH [LPRINT]**size**

FORMAT 2 WIDTH **#filenum,size**

FORMAT 3 WIDTH **device,size**

PURPOSE Sets the line width for the screen or printer in number of characters.

■ ■ ■

size is a number from 0 to 255. After **size** characters have been output, GWBASIC will insert a carriage return/line feed sequence before the next character is processed.

#filenum is a number from 1 to 15, the number of the file that is opened.

device is SCRN:, LPTn:, or COMn:.

Format 1

This format without LPRINT sets the screen width (as does WIDTH "SCRN:",**size**). For 325-line systems **size** is 80, the default. For 400-line systems during emulation modes 1 and 2 (see SCREEN statement) and for systems with color/graphics monitor adapters, **size** may be either 40 or 80 columns.

NOTE: For 400-line systems during emulation modes 1 and 2 (see SCREEN statement) and for systems with a color/graphics adapter, WIDTH 80 following SCREEN 1 forces the screen into high resolution, and WIDTH 40 following SCREEN 2 forces the screen into medium resolution.

This format with LPRINT sets the line width for the printer.

Format 2

WIDTH **#filenum,size** allows you to change the width of the device associated with **#filenum** any time the file is open. Note that the file must first be opened.

Format 3

WIDTH **device,size** stores a width assignment without changing the current setting. It can be used for LPT1:, LPT2:, COM1:, COM2:, COM3: and COM4:. A subsequent OPEN statement will use the new value. If the device is already open, the width will not change immediately.

LPRINT, LLIST, and LIST,"LPTn:" are all affected by this statement.

Out-of-range and illegal values result in an "Illegal function call" error, and the previous value is retained.

NOTE: For communications files, changing the width does not alter either the receive or transmit buffer, size or contents.

```
10 LPRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
10 WIDTH "LPT1:",18
20 LPRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
 ABCDEFGHIJKLMNOPQR
 STUVWXYZ
Ok
```

In the preceding example, output is directed to the printer with the LPRINT statement. Changing the width causes all output lines to be truncated to eighteen characters.

WINDOW Statement

FORMAT WINDOW [[SCREEN] (x1,y1)-(x2,y2)]

PURPOSE Redefines the dimensions of the current viewport for subsequent graphics statements.

■ ■ ■

This statement can only be used in graphics mode (SCREEN 1, 2, 104 or 105).

(x1,y1)-(x2,y2) are the "world coordinates", used to define the new dimensions of the viewport.

WINDOW allows you to draw lines, graphs, or objects in space not bounded by the physical dimensions of the screen. This is done by using programmer-defined coordinates called "world coordinates". A world coordinate is any valid pair of single-precision floating-point numbers. When you have redefined the viewport in this way, subsequent graphics statements will scale their output to the world coordinates.

BASIC 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 screen, BASIC has to know what portion of the world coordinate space contains the information to be displayed. This rectangular region in world coordinate space is called a "window".

RUN, or WINDOW with no parameters, disables this window transformation.

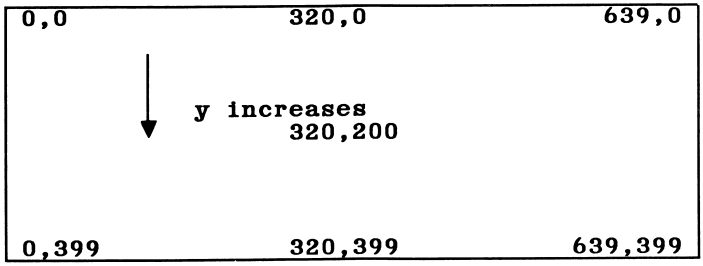
WINDOW inverts the y-axis of the world coordinates so that screen coordinates coincide with the traditional Cartesian arrangement: x increases left to right, and y decreases top to bottom.

The SCREEN option does not invert the y-coordinate.

To illustrate the use of this statement, assume that a program has executed:

```
NEW
SCREEN 105:CLS
```

The screen appears as:

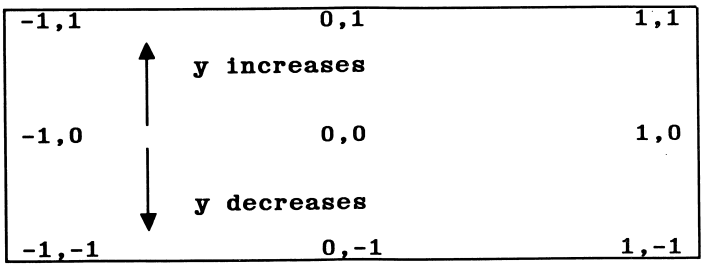


NOTE: The above figure relates to 400-line screens. For 325-line screens, substitute 324 for 399 and 320,160 for 320,200.

If you now execute:

```
WINDOW (-1,-1)(1,1)
```

the screen appears as:



The y-coordinate has been inverted, so **(x1,y1)** refers to the bottom left of the viewport and **(x2,y2)** to the top right. This allows the screen to be viewed in true Cartesian coordinates. Thus in the above example, the statement:

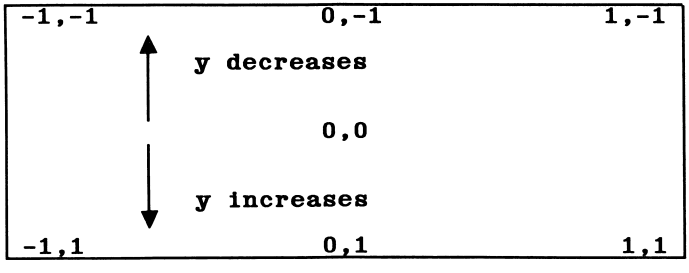
```
LINE (-1,1) - (1,-1),1
```

would draw a line from the top left to the bottom right of the screen.

If you use the SCREEN option and execute:

```
WINDOW SCREEN (-1,-1)(1,1)
```

the screen appears as:



Here the y-coordinate is not inverted, so **(x1,y1)** refers to the top left and **(x2,y2)** to the bottom right. Thus, following the WINDOW SCREEN statement above, the statement:

```
LINE (0,0) - (1,1),1
```

would draw a line from the center to the bottom right of the screen.

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"& the same line on a
    redefined window"
```

WRITE Statement

FORMAT WRITE **[list of expressions]**

PURPOSE Outputs data on the screen.

■ ■ ■

list of expressions consists of numeric and/or string expressions, separated by commas or semicolons.

If **list of expressions** is omitted, a blank line is output.

When the items are output, each is separated from the last by a comma. Strings are delimited by quotation marks. After the last item in the list is printed, GWBASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement. The only differences between the two are that WRITE inserts commas between the items, delimits strings with quotation marks, and does not precede positive numbers by blanks.

```
10 A=80: B=90: C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80,90,"THAT'S ALL"
Ok
```

Here A and B are numeric values, while C$ is a string.

WRITE # Statement

FORMAT WRITE #**filenum,list of expressions**

PURPOSE Writes data to a sequential file.

■ ■ ■

filenum is the number under which the file was opened in output (O) mode.

list of expressions contains string or numeric expressions separated by commas.

WRITE # and PRINT # are similar except that WRITE # places commas between the items as they are written, encloses strings with quotation marks, and does not put a blank in front of a positive number.

A carriage return/line feed sequence is inserted after the last item in the list is written.

For an example Let A$="HELLO" and B$="FOLKS". The statement:

```
WRITE #1,A$,B$
```

writes the following image to the file:

```
"HELLO","FOLKS"
```

A subsequent INPUT # statement, such as

```
INPUT #1,A$,B$
```

would input "HELLO" to A$ and "FOLKS" to B$.

Section 8

USING ASSEMBLY LANGUAGE SUBROUTINES

You can call assembly language subroutines from your GWBASIC program with the CALL or CALLS statement, or the USR function.

We recommend that you use the CALL or CALLS statement for interfacing assembly language programs with GWBASIC. These statements are more readable and can pass multiple arguments. In addition, the CALL statement is compatible with more languages than its alternative, the USR function.

MEMORY ALLOCATION

Memory space must be set aside for an assembly language subroutine before it can be loaded. To do so, use the /M: switch of the GWBASIC command during startup. The /M: switch sets the highest memory location to be used by GWBASIC.

In addition to the GWBASIC code area, GWBASIC uses up to 64K of memory beginning at its data segment (DS) - see the memory map in Appendix G.

If more stack space is needed when an assembly language subroutine is called, you can save the GWBASIC stack and set up a new stack for use by the assembly language subroutine. The GWBASIC stack must be restored, however, before you return from the subroutine.

LOADING AN ASSEMBLY LANGUAGE PROGRAM INTO MEMORY

You can load the assembly language subroutine into memory in several ways, the most simple being to use the BLOAD command (see Section 7). Alternatively, you could SHELL a program that exits, but stays resident, leaving the linked, relocated image in memory (see the SHELL statement). As a third choice, you could execute a program that exits but stays resident, and then run BASIC.

The following guidelines must be observed if you choose to BLOAD, or read and poke, an EXE file into memory:

1. Make sure the subroutines do not contain any long references, or address offsets that exceed 64K or that take the user out of the code segment. These long references require handling by the EXE loader.

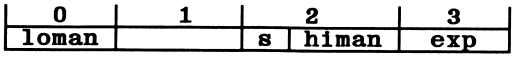
2. Skip over the first 512 bytes (the header) of the linker's output file (EXE), then read in the rest of the file.

INTERNAL REPRESENTATION OF NUMBERS

The following section describes the internal representation of numbers in GWBASIC. Knowledge of these arrangements is critical for many assembly language programming routines.

Single Precision - 24 bit mantissa

Single precision numbers are represented as follows:



where

loman = the low mantissa
s = the sign
himan = the high mantissa
exp = the exponent

If **exp** = 0, then (the single-precision value) **number** = 0.

If **exp** <> 0, then the mantissa is normalized and:

```
number = sgn * 0.1man * 2 ** (exp80h)
```

where

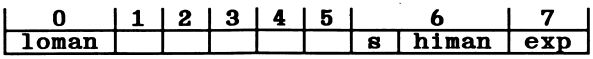
man = **himan** (without the **s** bit) through to **loman**

That is, in single precision (hex notation - bytes low to high):

```
05000080  =  0.5
05008080  = -0.5
```

Double Precision - 56 bit mantissa

Double precision numbers are represented as follows:



where

loman = the low mantissa
s = the sign
himan = the high mantissa
exp = the exponent

CALL STATEMENT

The CALL statement is the recommended way of interfacing assembly language subroutines with GWBASIC. Do not use the USR function unless you are running previously-written subroutines that already contain USR functions.

The syntax of the CALL statement is:

CALL **variable name** [(**argument list**)]

where **variable name** is the name of a numeric variable which contains an offset into the current segment. This address is the starting point in memory of the subroutine being called. The current segment is either the default, or that which has been defined by a DEF SEG statement.

argument list contains the variables or constants, separated by commas, that are to be passed to the subroutine.

Invoking the CALL statement causes the following to occur:

1. For each argument in the argument list, the two-byte offset of the argument's location within the BASIC segment is pushed onto the stack.

2. Control is transferred to the subroutine with an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in **variable name**.

Figures 8-1 and 8-2 illustrate the state of the stack at the time the CALL statement is executed, and the condition of the stack during execution of the called subroutine.

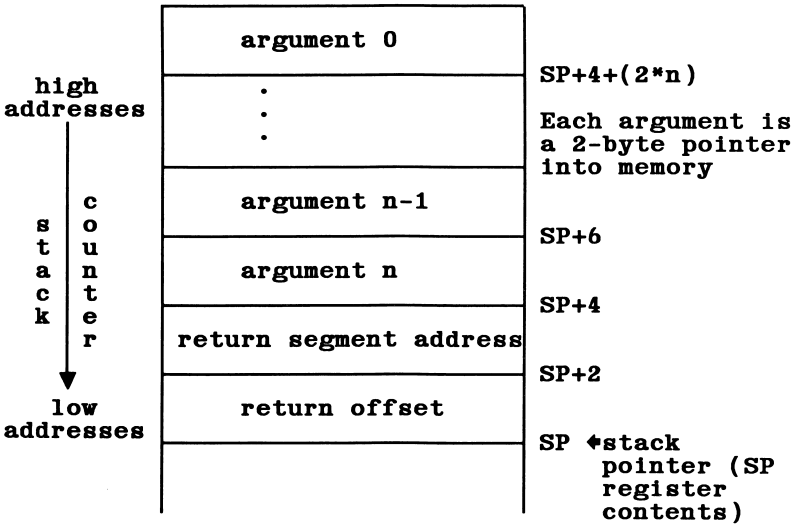


Figure 8-1. Stack Layout When CALL Statement is Activated

After the CALL statement has been activated, the subroutine has control. Arguments may be referenced by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP.

Observe the following rules when coding a subroutine:

1. The called routine must preserve segment registers DS, ES, SS, and the base pointer (BP). If interrupts are disabled in the routine, they must be enabled before exiting. The stack must be cleaned up on exit.

2. The called program must know the number and length of the arguments passed. The following routine shows an easy way to reference arguments:

```
PUSH  BP
MOV   BP,SP
ADD   BP, (2*number of arguments)+4
```

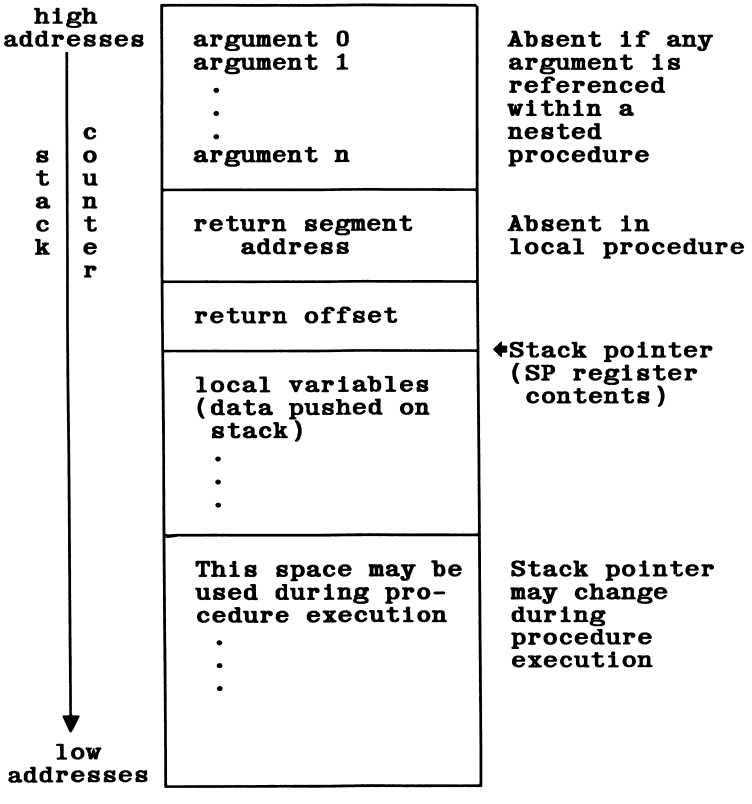


Figure 8-2. Stack Layout During Execution of a CALL Statement

Then:

argument 0 is at BP
argument 1 is at BP-2
argument n is at BP-2*n

(number of arguments = n+1)

3. Variables may be allocated either in the code segment or on the stack. Be careful not to modify the return segment and offset stored on the stack.

4. The called subroutine must clean up the stack. A preferred way to do this is to perform a RET **n** statement (where **n** is two times the number of arguments in the argument list) to adjust the stack to the start of the calling sequence.

5. Values are returned to GWBASIC by including in the argument list the name of the variable that will receive the result. Details of the internal representation of numbers in GWBASIC are given above.

6. If the argument is a string, the argument's offset points to 3 bytes which, as a unit, are called the **string descriptor**. Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

CAUTION

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. For example, use:

```
20 A$ = "BASIC"+""
```

This will force the string literal to be copied into string space. Then the string may be modified without affecting the program.

7. The contents of a string may be altered by user routines, but the descriptor must not be changed.

   Do not write past the end-of-string. GWBASIC cannot correctly manipulate strings if their lengths are modified by external routines.

8. Data areas needed by the routine must be allocated either in the CODE segment of the user routine or on the stack. It is not possible to declare a separate data area in the user assembler routine.

Example of CALL statement:

```
100 DEF SEG=&H8000
110 FOO=&H7FA
120 CALL FOO(A,B$,C)
      .
      .
      .
```

Line 100 sets the segment to 8000 hex. The value of variable FOO is added into the address as the low word after the DEF SEG value is left-shifted 4 bits. Here, the long call to FOO will execute the subroutine at location 8000:7FA hex (absolute address 807FA hex).

The following sequence in assembly language demonstrates access to the arguments passed. The returned result is stored in the variable C.

```
PUSH    BP          ;Set up pointer to arguments
MOV     BP,SP
ADD     BP,(4+2*3)
MOV     BX,[BP-2]   ;Get address of B$ descriptor
MOV     CL,[BX]     ;Get length of B$ in CL
MOV     DX,1[BX]    ;Get addr of B$ text in DX
          .
          .
          .
MOV     SI,[BP]     ;Get address of 'A' in SI
MOV     DI[BP-4]    ;Get pointer to 'C' in DI
MOVS    WORD        ;Store variable 'A' in 'C'
POP     BP          ;Restore pointer
RET     6           ;Restore stack, return
```

IMPORTANT: The called program must know the variable type for the numeric arguments passed. In the previous example, the instruction

```
MOVS WORD
```

will copy only two bytes. This is fine if variables A and C are integer. You would have to copy four bytes if the variables were single precision format and copy 8 bytes if they were double precision.

CALLS STATEMENT

The CALLS statement should be used to access subroutines that were written using FORTRAN calling conventions. CALLS works just like CALL, except that with CALLS the arguments are passed as segmented addresses, rather than as unsegmented addresses.

Because FORTRAN routines need to know the segment value for each argument passed, the segment is pushed and then the offset is also pushed. For each argument, four bytes are pushed rather than 2, as in the CALL statement. Therefore, if your assembler routine uses the CALLS statement, **n** in the RET **n** statement is four times the number of arguments.

USR FUNCTION

Although using the CALL statement is the recommended way of calling assembly language subroutines, the USR function is also available for this purpose. This ensures compatibility with older programs that contain USR functions.

```
USR[digit][(argument)]
```

where **digit** is from 0 to 9. **digit** specifies which USR routine is being called. If **digit** is omitted, USR0 is assumed.

argument is any numeric or string expression. Arguments are discussed in detail in the following paragraphs.

A DEF SEG statement **must** be executed prior to a USR function call to assure that the code segment points to the subroutine being called. The

segment address given in the DEF SEG statement determines the starting segment of the subroutine.

For each USR function, a corresponding DEF USR statement must be executed to define the USR function call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register AL contains a value that specifies the type of argument that was given. The value in AL may be one of the following:

Value in AL	Type of argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating-point number
8	Double precision floating-point number

If the argument is a number, the BX register points to the floating-point accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-2 contains the upper 8 bits of the integer.
FAC-3 contains the lower 8 bits of the integer.

For versions of GWBASIC that use binary floating-point:

FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive, 1 = negative).

If the argument is a single precision floating-point number:

FAC-2 contains the middle 8 bits of mantissa.
FAC-3 contains the lowest 8 bits of mantissa.

If the argument is a double precision floating-point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the DX register points to 3 bytes which, as a unit, are called the **string descriptor**. Byte 0 of the string descriptor contains the length of the string (0 to 255 characters). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in the GWBASIC data segment.

CAUTION

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy the program this way.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

GWBASIC has extended the USR function interface to allow calls to MAKINT and FRCINT. This allows access to these routines without giving their absolute addresses. The address ES:BP is used as an indirect far pointer to the routines FRCINT and MAKINT.

To call FRCINT from a USR routine use CALL DWORD ES:[BP].

To call MAKINT from a USR routine use CALL DWORD ES:[BP+4].

Example:

```
110 DEF USR0=&H8000 'Assumes decimal
                    argument /M:32767
120 X=5
130 Y = USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with that of the argument used.

Appendix A
SEQUENTIAL AND RANDOM FILES

There are two types of data files that may be created and accessed by a GWBASIC program: sequential and random.

SEQUENTIAL FILES

Sequential files are easier to create than random files, but they are limited in flexibility and speed when it comes to accessing data. The data (ASCII characters) written to a sequential file is stored one item after another in the order sent. The data is read back in the same way.

Statements and functions used with sequential files are:

CLOSE	LOF
EOF	OPEN
INPUT#	PRINT #
INPUT$	PRINT # USING
LINE INPUT#	WIDTH
LOC	WRITE #

Creating and Accessing a Sequential File

The following program steps are required to create a file and access the data in it.

1. Use OPEN to open the file for output or append.
2. Write data to the file using the WRITE #, PRINT #, or PRINT # USING statements.
3. To access the data in the file, first close the file (using CLOSE) and then reopen it for input (using OPEN).
4. Read data from the file into the program by using the INPUT # or LINE INPUT # statements.

Following is a short program that creates a sequential file, "DATA", on the default directory from information you input at the terminal.

```
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? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

The next program accesses the file created above and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
Ok
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

The program above reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To

avoid getting this error, insert line 15 below, which uses the EOF function to test for end-of-file.

```
15 IF EOF(1) THEN END
```

Then change line 40 to GOTO 15. Since the end of file is indicated by a special character in the file with ASCII code 26 (hex 1A), you should not put a CHR$(26) in a sequential file.

A program that creates a sequential file can also write formatted data to disk with the PRINT # USING statement. For example, the statement

```
PRINT #1,USING "####.##,";A,B,C,D
```

could 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.

Note that the LOC function, when used with a sequential file, returns the number of records that have been written to or read from the file since it was opened. (A record is a 128-byte block of data.) The LOF function returns the number of bytes allocated to the file. For example,

```
100 IF LOC(1)>50 THEN STOP
```

would end program execution if more than 50 sectors had been written to or read from file #1 since it was opened.

Adding Data to a Sequential File

If you have a sequential file on disk and want to add more data to the end of it, you cannot simply open the file for output mode and start writing data because as soon as you do this, you destroy the file's content. Instead, open the file for APPEND. See OPEN statement for further information.

RANDOM FILES

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. They require less room on the disk because GWBASIC stores them in a packed binary format. (Recall that a sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files, though, is that data can be accessed randomly, i.e., anywhere on the disk. It is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records, and each record is numbered. The maximum length a record may have is 32767 bytes.

Statements and functions used with random files are:

CLOSE	LOF
CVD	MKD$
CVI	MKI$
CVS	MKS$
FIELD	OPEN
GET	PUT
LSET/RSET	WIDTH
LOC	

Creating a Random File

Use the following program steps to create a random file.

1. Open the file for random access.

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

3. Use LSET or RSET to move the data into the random buffer. Numeric values must be converted into string values when placed in the buffer. To do this, use the following functions: MKI$ for integers, MKS$ for single-precision values, and MKD$ for double-precision values.

4. Use PUT to write the data from the buffer to the disk.

The following program writes information entered at the keyboard to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2DIGIT 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
```

Do not use a string variable defined in a FIELD statement in an input statement or on the left side of an assignment (LET) statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

Accessing a Random File

Use the following steps to access a random file.

1. Open the file for random access.

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

   Note that in a program that performs both 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.

Note that when you close the file, the variable assigned to the GET statement is no longer accessible.

The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers. This is done with the "convert" functions: CVI for integers, CVS for single-precision values, and CVD for double-precision values.

The following program accesses the random file named "FILE" that was created above. When the two-digit code is entered at the keyboard the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2DIGIT CODE";CODE%
40 GET #1, CODE%
```

```
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

The LOC function, with random files, returns the "current record number." The current record number is the last number used in a GET or PUT statement. For example, the statement:

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file #1 is higher than 50.

A Sample Program

Following is an inventory program that illustrates random file access. In this program the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900 through 960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 140 through 210 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
120 OPEN"R",#1,"INVEN.DAT",39
130 FIELD #1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 A
    P$
140 PRINT:PRINT "FUNCTIONS:":PRINT
150 PRINT "1,INITIALIZE FILE"
160 PRINT "2,CREATE A NEW ENTRY"
170 PRINT "3,DISPLAY INVENTORY FOR ONE PART"
180 PRINT "4,ADD TO STOCK"
190 PRINT "5,SUBTRACT FROM STOCK"
200 PRINT "6,DISPLAY ALL ITEMS BELOW REORDER
    LEVEL"
210 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
220 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
    "BAD FUNCTION NUMBER":GOTO 140
```

```
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 210
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%;"INSTOCK":
    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 REM 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$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART
    NUMBER": GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

Appendix B
ADVANCED GRAPHICS INFORMATION

CONFIGURING YOUR COMPUTER FOR GWBASIC AND GRAPHICS: 325-LINE DESKTOP USERS

Inside your computer along the front edge is a jumper area labeled "B". For GWBASIC to use high-resolution graphics properly, this area must be jumpered pin 2 to pin 4. Jumpering these pins selects a graphics area in the first 256K of memory. Note that this is the normal configuration. Jumpering pins 1 and 3 selects graphics in the second 256K. GWBASIC will use graphics properly only with graphics available in the first 256K.

NOTE: Users with portable 325-line PCs or any 400-line system do not need to perform this procedure.

ADVANCED INFORMATION FOR ASSEMBLY LANGUAGE PROGRAMMERS

It is recommended that you use caution when programming in assembly language. In addition, because the information included in this document does not address the topic of hardware, you should refer to the appropriate Technical Reference manual for detailed hardware information.

The formula for calculating an address and mask for the monochrome graphics screen is as follows. Note that the calculation for 400-line screen emulation modes 1 and 2 is given separately (see "SCREEN" statement for further information on screen modes).

```
ADDRESS = 2048*MOD(Y,SCAN)+[80*INT(Y/SCAN)]+
          [INT(X/8)]
```

where

SCAN= 13 for 325-line systems
16 for 400-line systems

```
Z = MOD(X,8)
MASK = 2^n
```

where

n = **Z** for 325-line systems
(7-Z) for 400-line systems

For 400-line graphics emulation modes:

```
ADDRESS = Y*40+8192*MOD(Y/2)+INT(X/RES)

MASK = 2^[(RES-1)-MOD(X,RES)]
```

where

RES= **4** for medium-resolution graphics emulation mode
8 for high-resolution graphics emulation mode

In the following algorithms ES is the memory segment where the graphics screen resides, BX is the address of the pixel, and AL is the pixel mask. Use these to:

Set a point:

```
OR    ES:[BX],AL     ;Set bit in graphics screen
```

Erase a point:

```
NOT   AL             ;Reverse bit mask
AND   ES:[BX],AL     ;Mask off pixel
```

Toggle a point:

```
XOR   ES:[BX],AL     ;Toggle pixel
```

Test a point:

```
AND   AL,ES:[BX]
JNZ   Pixel Set      ;Point is on
...                  ;Point is off
```

The following assembly language routines calculate an address and mask for setting or resetting a point. The X position should be passed in the CX

register and the Y position in the DX register. The address is returned in the BX register and the mask in the AL register. The first routine may be used with 325-line systems; the second routine may be used with 400-line machines, except during emulation (modes 1 and 2); and the third routine may be used with 400-line systems during emulation.

Assembly language routine for 325-line systems:

```
MAPXYC PROC
       PUSH    CX
       PUSH    BP
       PUSH    ES
       MOV     AX,DRAWING_SEG ;(Memory segment
       MOV     ES,AX          ;where the graphics
       MOV     AX,DX          ;screen is located)
       MOV     BL,13
       DIV     BL
       MOV     BH,AL
       AND     BX,OFFOOH
       ROR     BX,1
       ROR     BX,1
       MOV     BP,BX
       ROR     BP,1
       ROR     BP,1
       ADD     BX,BP
       AND     AX,OFOOH
       ROL     AX,1
       ROL     AX,1
       ROL     AX,1
       ADD     BX,AX
       MOV     AX,CX
       SHR     AX,1
       SHR     AX,1
       SHR     AX,1
       ADD     BX,AX
       MOV     AL,128
       AND     CL,7
       INC     CL
       ROL     AL,CL
       POP     ES
       POP     BP
       POP     CX
       RET
MAPXYC ENDP
```

Assembly language routine for 400-line systems (nonemulation modes):

```
MAPXYC PROC
       MOV     AX,DX
       AND     DX,15
       SUB     AX,DX
       ROR     DX,1
       ROR     DX,1
       ROR     DX,1
       ROR     DX,1
       ROR     DX,1
       ADD     DX,AX
       SHL     AX,1
       SHL     AX,1
       ADD     AX,DX
       MOV     BX,AX
       MOV     AX,CX
       SHR     CX,1
       SHR     CX,1
       SHR     CX,1
       ADD     BX,CX
       XCHG    AX,CX
       MOV     AL,128
       AND     CL,7
       ROR     AL,CL
       RET
MAPXYC ENDP
```

Assembly language routine for 400-line systems (emulation modes):

```
MAPXYC PROC
       PUSH    BP
       XOR     BX,BX
       SHR     DX,1
       JNC     C1_MAP
       MOV     BX,8192
C1_MAP:
       XCHG    DH,DL
       SHR     DX,1
       SHR     DX,1
       MOV     AX,DX
       SHR     AX,1
       SHR     AX,1
       ADD     AX,DX
       ADD     AX,BX
       MOV     BP,AX
```

```
        MOV     BX,CX
        SHR     CX,1
        SHR     CX,1
        CMP     "mode","medium res"
                (see note)
        JE      C1_MED
;
;HIGHRES CALCULATION (640x200)
;
        SHR     CX,1
        XCHG    BX,CX
        AND     CL,7
        MOV     AL,128
        JMP     C1_RET
;
;MEDIUMRES CALCULATION (320x200)
;
C1_MED:
        XCHG    BX,CX
        MOV     AL,192
        AND     CL,3
        ADD     CL,CL
C1_RET:
        ROR     AL,CL
        ADD     BX,BP
        POP     BP
        RET
MAPXYC ENDP
```

NOTE: You must define a method of indicating which graphics mode is currently operative. For example, the mode number may be stored so that 1 indicates medium-resolution graphics and 2 indicates high resolution (see the appropriate routine line under C1 MAP"). Alternatively, a unique mapping routine can be defined for each mode.

The following routines set the 6845 video controller chip to point to the graphics display segment addressed by DRAWING_SEG, and are for 325- and 400-line machines respectively. Note that if you want the cursor to remain on the screen, you will have to write your own cursor position routine. The Cursor High register (hex 0E) in this chip needs to point to the same segment of memory as the Start_High register (hex 0C), which always points to the graphics screen. Refer to a 6845 data sheet for more information on programming this chip.

CAUTION

Programming of the video chip is not recommended practice. If it is necessary, take great care as incorrect programming can cause unpredictable results throughout the system.

The routine for 325-line systems is:

```
GRDISP PROC
       MOV     DX,3B4H          ;DX=Address of 6845
       MOV     CX,DRAWING_SEG   ;CX=Graphics
                                 segment

       MOV     AH,CH            ;Graphics segment
                                 high

       MOV     AL,0CH           ;6845 register 0Ch
       OUT     DX,AL            ;Select register
                                 0Ch

       MOV     AL,AH
       INC     DX
       OUT     DX,AL            ;Set Reg.0Ch =
                                 GrSegHigh
       RET
GRDISP ENDP
```

The routine for 400-line systems is:

```
GPDISP PROC
       MOV     BX, DRAWING_SEG
       MOV     AH, BH
       AND     AH, 60H                  ;Extract
                                         128K block
       MOV     CL, 3
```

```
        SHR     AH, CL          ;Mark 128K block to
                                ;position for system
                                ;control port 3BF,
                                ;bits M0 and M1
        MOV     DX, 3BFH        ;System control port
        IN      AL, DX
        AND     AL, 11110011B   ;Zero out current
                                ;128K block
        OR      AL, AH          ;OR initiates new
                                ;block
        OUT     DX, AL
        AND     BX, 1FFFH       ;BX:=offset of
                                ;drawing segment
                                ;within 128K block
        MOV     DX, 3B4H
        MOV     AL, 0CH
        OUT     DX, AL          ;Select register 12
        INC     DX
        MOV     AL, BH
        OUT     DX, AL          ;Send high byte
                                ;of segment to
                                ;register 12
        DEC     DX
        MOV     AH, 0DH
        OUT     DX, AL          ;Select register 13
        INC     DX
        MOV     AL, BL
        OUT     DX, AL          ;Send low byte
                                ;of segment to
                                ;register 13
        RET
GPDISP  ENDP
```

You will also need to enable the type of display you want to use: graphics only, text only, or mixed text and graphics.

To enable graphics display only:

```
MOV     DX,3B8H
MOV     AL,0A0H
OUT     DX,AL
```

To enable text display only:

```
MOV     DX,3B8H
MOV     AL,28H
OUT     DX,AL
```

To enable both graphics and text:

```
MOV     DX,3B8H
MOV     AL,0A8H
OUT     DX,AL
```

GRAPHICS MEMORY MAP

Graphics memory resides in a 32K boundary that you set aside when you specify the active page (see the SCREEN statement). On 325-line systems, it consists of 325 lines of 80 bytes per line. On 400-line systems, it consists of 400 lines of 80 bytes per line. Each byte contains 8 pixels. This gives a resolution of 640 x 325 pixels for 325-line systems and 640 x 400 pixels for 400-line systems.

On both 325-line and 400-line systems, one line on the screen is created by scan lines from individual 2K blocks of memory (except for the 400-line emulation modes 1 and 2 as referenced in the note below). For 325-line systems, 13 scan lines make up each line on the screen (1 scan line from each of the 13 blocks of memory). For 400-line systems, 16 scan lines make up each line on the screen (1 scan line from each of the 16 blocks of memory). See Figure B-1.

The offset within each 2K memory block can be calculated by the following formula:

(screen line number - 1) * width (80 bytes)

For example, the offset for screen line 2 is 80:

(2-1) * 80 = 80

NOTE: The process is completely different during 400-line graphics emulation (modes 1 and 2). For information on the internal handling of the graphics emulation modes, see the 400-line Technical Reference manual.

Figures B-2 and B-3 illustrate graphics memory layouts for the 325-line system and the 400-line system (except for emulation modes), respectively.

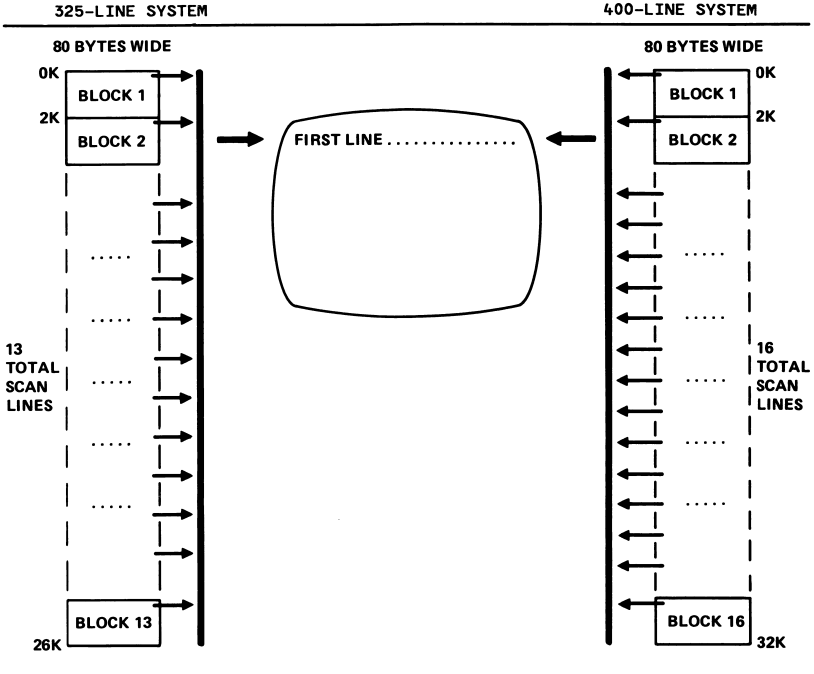


Figure B-1. Graphics Display Configuration

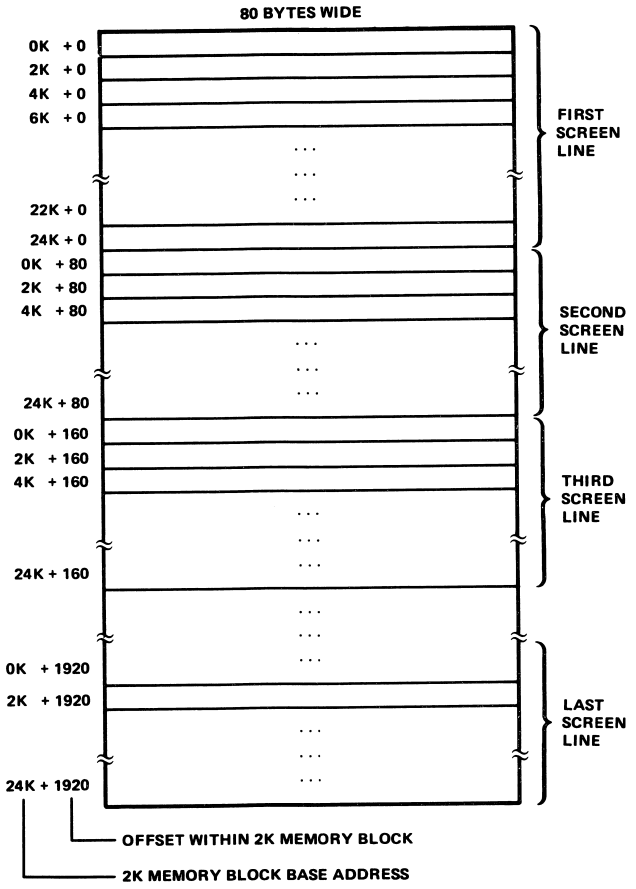


Figure B-2. 325-Line Graphics Memory Map

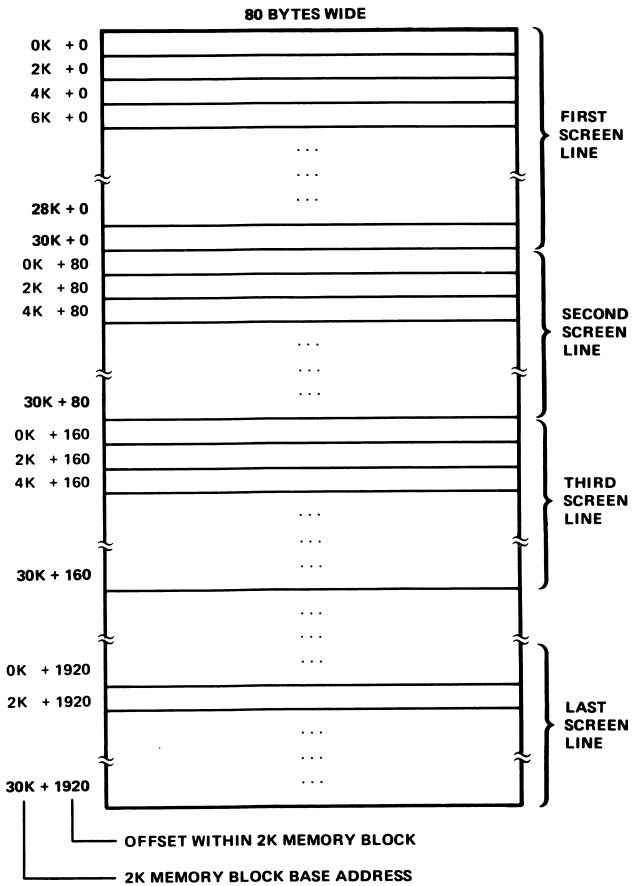


Figure B-3. 400-Line Graphics Memory Map

Appendix C
ASCII CHARACTER CODES

Table C-1 provides for ASCII codes 000-255 the associated hexadecimal value (Hex) and character (CHR).

Use Alt and the numeric keypad to enter these characters.

Refer to Section 2 and "Characters" in Section 5 for information on additional key combinations.

Table C-1

ASCII CHARACTER CODES

Decimal	Hex	CHR	Decimal	Hex	CHR
000	00	NUL	036	24	$
001	01	SOH	037	25	%
002	02	STX	038	26	&
003	03	ETX	039	27	'
004	04	EOT	040	28	(
005	05	ENQ	041	29	)
006	06	ACK	042	2A	*
007	07	BEL	043	2B	+
008	08	BS	044	2C	,
009	09	HT	045	2D	-
010	0A	LF	046	2E	.
011	0B	VT	047	2F	/
012	0C	FF	048	30	0
013	0D	CR	049	31	1
014	0E	SO	050	32	2
015	0F	SI	051	33	3
016	10	DLE	052	34	4
017	11	DC1	053	35	5
018	12	DC2	054	36	6
019	13	DC3	055	37	7
020	14	DC4	056	38	8
021	15	NAK	057	39	9
022	16	SYN	058	3A	:
023	17	ETB	059	3B	;
024	18	CAN	060	3C	<
025	19	EM	061	3D	=
026	1A	SUB	062	3E	>
027	1B	ESC	063	3F	?
028	1C	FS	064	40	@
029	1D	GS	065	41	A
030	1E	RS	066	42	B
031	1F	US	067	43	C
032	20	SPC	068	44	D
033	21	!	069	45	E
034	22	"	070	46	F
035	23	#	071	47	G

Table C-1 (Cont.)

ASCII CHARACTER CODES

Decimal	Hex	CHR	Decimal	Hex	CHR
072	48	H	100	64	d
073	49	I	101	65	e
074	4A	J	102	66	f
075	4B	K	103	67	g
076	4C	L	104	68	h
077	4D	M	105	69	i
078	4E	N	106	6A	j
079	4F	O	107	6B	k
080	50	P	108	6C	l
081	51	Q	109	6D	m
082	52	R	110	6E	n
083	53	S	111	6F	o
084	54	T	112	70	p
085	55	U	113	71	q
086	56	V	114	72	r
087	57	W	115	73	s
088	58	X	116	74	t
089	59	Y	117	75	u
090	5A	Z	118	76	v
091	5B	[	119	77	w
092	5C	\	120	78	x
093	5D	]	121	79	y
094	5E	^	122	7A	z
095	5F	_	123	7B	{
096	60	`	124	7C	|
097	61	a	125	7D	}
098	62	b	126	7E	~
099	63	c	127	7F	DEL

Table C-1 (Cont.)

ASCII CHARACTER CODES

Decimal	Hex	Character
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	Pt
159	9F	ƒ

Decimal	Hex	Character
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	▒
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Table C-1 (Cont.)

ASCII CHARACTER CODES

Decimal	Hex	Character
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	■
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Decimal	Hex	Character
224	E0	α
225	E1	ß
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)

EXTENDED CODES

Certain key combinations return a two-character value for the INKEY$ function. The first character is a null (CHR$(0)), and the second is as shown in the table below.

Table C-2

EXTENDED KEY CODES

A. FUNCTION KEYS (when disabled)				
Key	Normal	Shift	Ctrl	Alt
F1	59	84	94	104
F2	60	85	95	105
F3	61	86	96	106
F4	62	87	97	107
F5	63	88	98	108
F6	64	89	99	109
F7	65	90	100	110
F8	66	91	101	111
F9	67	92	102	112
F10	68	93	103	113

B. OTHER KEYS			
Key	Normal	Shift	Ctrl
@			3
Tab		15	
Home	71		119
Up	72		
Pg Up	73		132
Left	75		115
Right	77		116
End	79		117
Down	80		
Pg Dn	81		118
Ins	82		
Del	83		
PrtSc			114

Table C-2 (Cont.)

C. ALPHANUMERIC KEYS					
Key	Alt	Key	Alt	Key	Alt
A	30	N	49	1	120
B	48	O	24	2	121
C	46	P	25	3	122
D	32	Q	16	4	123
E	18	R	19	5	124
F	33	S	31	6	125
G	34	T	20	7	126
H	35	U	22	8	127
I	23	V	47	9	128
J	36	W	17	0	129
K	37	X	45	-	130
L	38	Y	21	=	131
M	50	Z	44		

Appendix D
LIST OF GWBASIC RESERVED WORDS

```
ABS
AND
ASC
ATN
AUTO

BEEP
BLOAD
BSAVE

CALL
CDBL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COLOR
COM
COMMON
CONT
COS
CSNG
CSRLIN
CVD
CVI
CVS

DATA
DATE$
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DEF FN
DEF USR
DELETE
DIM
DRAW

EDIT
ELSE
END
ENVIRON
EOF
ERASE
ERDEV
ERL
ERR
ERROR
END
EXP

FIELD
FILES
FIX
FOR
FRE

GET
GOSUB
GOTO

HEX$

IF
IMP
INKEY$
INP
INPUT
INPUT#
INPUT$
INSTR
INT
IOCTL

KEY
KILL

LCOPY
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET

MERGE
MID$
MKD$
MKI$
MKS$
MKDIR
MOD

NAME
NEW
NEXT
NOT

OCT$
OFF
ON
OPEN
OPEN COM
OPTION
OR
OUT

PAINT
PEEK
PEN
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PRINT#
PSET
PUT

RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RMDIR
RND
RSET
RUN

SAVE
SCREEN
SGN
```

SHELL	STRING$	TRON	WEND
SIN	SWAP		WHILE
SOUND	SYSTEM	USING	WIDTH
SPACE		USR	WINDOW
SPC	TAB		WRITE
SQR	TAN	VAL	WRITE#
STEP	THEN	VARPTR	
STICK	TIME$	VARPTR$	XOR
STOP	TIMER	VIEW	
STR$	TO		
STRIG	TROFF	WAIT	

Appendix E
TRIGONOMETRIC FUNCTIONS

Trigonometric functions can be calucated in GWBASIC using the following formulae.

Table E-1

Trigonometric Functions

Function	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)+EXP(-X))
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)-EXP(-X))
Inverse Hyperbolic Sine	ARCSINH(X)=LOG(X+SQR(X*X+1))

Table E-1 (Cont.)

Function	Equivalent
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
Inverse Hyperbolic Cosine	ARCCOSH(X)=LOG(X+SQR(X*X-1))
Inverse Hyperbolic Tangent	ARCTANH(X)=LOG((1+X)/(1-X))/2

See DEF FN statement for further coding information.

Appendix F
SCAN CODES

The table below gives the keyboard scan codes. Each key on the keyboard is assigned a scan code, and it is this hexadecimal code which is used as the value of **scan** in the KEY statement when trapping user-defined keys.

Table F-1

KEYBOARD SCAN CODES

Key	Scan code	Key	Scan code	Key	Scan code
Esc	01	Ctrl	1D	Space	39
! 1	02	A	1E	CapsLk	3A
@ 2	03	S	1F	F1	3B
# 3	04	D	20	F2	3C
$ 4	05	F	21	F3	3D
% 5	06	G	22	F4	3E
^ 6	07	H	23	F5	3F
& 7	08	J	24	F6	40
* 8	09	K	25	F7	41
(9	0A	L	26	F8	42
) 0	0B	: ;	27	F9	43
_ -	0C	" '	28	F10	44
+ =	0D	~ `	29	NumLk	45
BSp	0E	Shf(L)	2A	ScrLk	46
Tab	0F	| \	2B	7 Home	47
Q	10	Z	2C	8 ↑	48
W	11	X	2D	9 PgUp	49
E	12	C	2E	-	4A
R	13	V	2F	4 ←	4B
T	14	B	30	5	4C
Y	15	N	31	6 →	4D
U	16	M	32	+	4E
I	17	< ,	33	1 End	4F
O	18	> .	34	2 ↓	50
P	19	? /	35	3 PgDn	51
{ [	1A	Shf(R)	36	0 Ins	52
}]	1B	PrtSc	37	Del	53
Rtn	1C	Alt	38		

Appendix G
TECHNICAL INFORMATION AND PROGRAMMING HINTS

CONTROL CODES

The table below lists the hexadecimal and decimal codes for the GWBASIC control characters and summarizes their functions.

Table G-1

GWBASIC CONTROL FUNCTIONS

Ctrl Key	Hex	Decimal	Function
Ctrl-A	01	001	Enter edit mode
Ctrl-B	02	002	Cursor to start of previous word
Ctrl-C	03	003	Break
Ctrl-E	05	005	Clear to end of line
Ctrl-F	06	006	Cursor to start of next word
Ctrl-G	07	007	Sound speaker
Ctrl-H	08	008	Destructive backspace
Ctrl-I	09	009	Tab (8 spaces)
Ctrl-J	0A	010	Line feed
Ctrl-K	0B	011	Cursor home
Ctrl-L	0C	012	Clear screen or viewport
Ctrl-M	0D	013	Carriage return
Ctrl-N	0E	014	Cursor to end of line
Ctrl-O	0F	015	Suspend/restart program output
Ctrl-Q	11	017	Restart suspended program
Ctrl-R	12	018	Toggle insert mode
Ctrl-S	13	019	Suspend program
Ctrl-T	14	020	Toggle function key display
Ctrl-U	15	021	Clear logical line
Ctrl-W	17	023	Delete word
Ctrl-X	18	024	Display previous program line

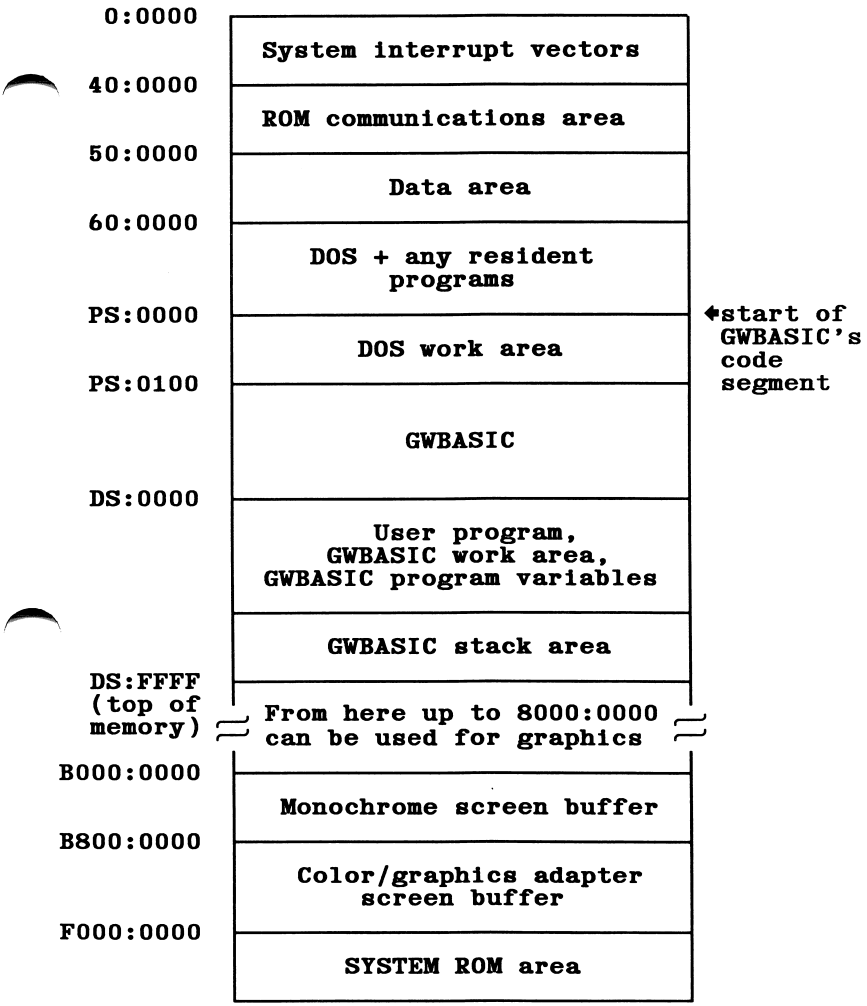
Table G-1 (Cont.)

Ctrl Key	Hex	Decimal	Function
Ctrl-Y	19	025	Display following program line
Ctrl-Z	1A	026	Clear to end of screen or window
Ctrl-\	1C	028	Cursor right
Ctrl-]	1D	029	Cursor left
Ctrl-^	1E	030	Cursor up
Ctrl-_	1F	031	Cursor down
None	FF	255	Mark line for deletion

The hex code FF (decimal 255) is a special code used by the screen editor. When a carriage return moves the cursor to a line that contains this code, the line is cleared. GWBASIC includes this code in the system messages it displays. Such messages may then be removed from the screen when they interfere with other material being displayed.

MEMORY MAP

Figure G-1 is a memory map for GWBASIC, showing whereabouts in memory the various sections of code reside. Addresses are in hexadecimal in the form segment:offset.



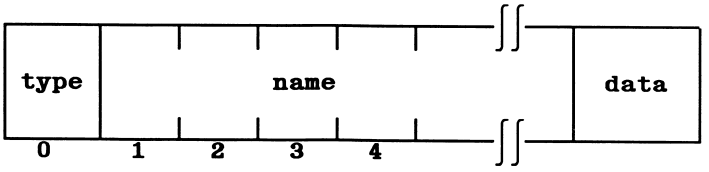
PS - DOS program segment
DS - GWBASIC data segment (work area)

GWBASIC stack size is 512 bytes or one-eighth of the available memory, whichever is smaller, unless given a different value by the CLEAR statement.

Figure G-1. Memory map for GWBASIC

HOW VARIABLES ARE STORED

Scalar variables are stored in GWBASIC's data area as follows:



type identifies the variable's type. The contents of byte 0 have the following meanings:

2 integer
3 string
4 single-precision
8 double-precision

name is the name of the variable. Bytes 1 and 2 hold the first two characters of the name. Byte 3 indicates the remaining number of characters in the variable name. They are stored starting at byte 4.

Since variable names take up at least three bytes, a one- or two-character name will occupy exactly three bytes. An x-character name will occupy x+1 bytes.

data follows the variable name. This parameter may be two, three, four, or eight bytes long according to its type. The VARPTR function returns the offset into the GWBASIC data segment of the first byte of this data.

For string variables, **data** is the string descriptor. The first byte of this descriptor contains the string length, in the range 0 to 255. The last two bytes contain the address of the string in GWBASIC's data space, which is the offset into the default segment.

Addresses are stored with the low byte first and the high byte second. This means that the second byte of the string descriptor contains the low byte of the offset, and the third byte contains the high byte of the offset.

For numeric variables **data** contains the actual value of the variable. Integer values occupy two bytes, with the low byte first and the high byte second. Single-precision values occupy four bytes in GWBASIC's internal floating-point binary format, while double-precision values occupy eight bytes in this format.

KEYBOARD BUFFER

Characters entered at the keyboard are stored in the keyboard buffer, which can hold up to 15 characters. Typing more than 15 characters will cause the speaker to beep.

INKEY$ reads one character from the keyboard buffer, while INPUT$ will read multiple characters. With INPUT$, if there are not the requested number of characters present, GWBASIC will wait until enough are typed.

You can clear the keyboard buffer with the following lines of code:

```
DEF SEG=0
POKE 1050,PEEK(1052)
```

This may be useful to do before calling for a user to "press any key".

GWBASIC's line buffer, where the program editor acts on characters received from keyboard buffer, is cleared with the following code:

```
DEF SEG:  POKE 106,0
```

SEARCH ORDER FOR PORTS

Printers associated with LPT1: and LPT2: are assigned when the computer is switched on. Printer ports are sought in a particular sequence. The first port found becomes LPT1: and the second LPT2:. The search order is as follows:

1. Built in parallel printer port
2. A parallel printer adapter modified to change its base address.

If the MODE command from DOS was used to reroute a printer, the change is effective in GWBASIC as well.

The communications devices COM1: to COM4: are assigned in the following order:

1. Built-in asynchronous communications adapter (COM1:)
2. Add-on asynchronous communications adapters (COM2: to COM4:)

SWITCHING DISPLAYS

GWBASIC will normally write to the color/graphics monitor adapter if present. Switch from one display to the other with the following code:

```
10 'switch to monochrome adapter
20 SCREEN 100
30 SCREEN 0
40 WIDTH 80

10 'switch to color adapter
20 SCREEN 101
30 SCREEN 0
40 WIDTH 40
```

Note that with this code the screen you are switching to is cleared. You may also need to keep track of the cursor location for each display.

SOME TECHNIQUES WITH A COLOR/GRAPHICS ADAPTER

Sixteen Background Colors

If you do not need blink, you can get all 16 colors for the background by including the following code in your GWBASIC program:

OUT &H3D8,8	in 40-column width
OUT &H3D8,9	in 80-column width

Character Color in Graphics Mode

When you display text characters while in graphics mode in medium resolution, the color of the characters is 3 and the background color is 0. To change the foreground color to 2 or 1 use:

```
DEF SEG: POKE &H4E, color
```

where **color** is the desired foreground color (note that color 0 is not allowed). Subsequent PRINT statements will use the specified color.

PROGRAMMING TECHNIQUES

General

1. Combine statements to take advantage of the 255-character line length. For example:

```
100 FOR I=1 TO 10
110 PRINT I,I^2
120 NEXT I
```

   can better be written

```
100 FOR I=1 TO 10: PRINT I,I^2: NEXT I
```

2. Avoid repetitive evaluation of expressions. When the identical calculation is necessary in several statements, evaluate the expression once and save the result in a variable. For example:

```
310 A=C4+XY
320 B=C4+X+Z
```

can more profitably be written

```
300 Q=C4+X
310 A=QY
320 B=Q+Z
```

Remember that assigning a constant to a variable will be faster than assigning the value of another variable to a variable.

3. Use simple arithmetic. Since addition is performed faster than multiplication, and multiplication is faster than division or exponentiation:

Instead of	Use
B=A/2	B=A*.5
B=A*2	B=A+A
B=A^3	B=A*A*A
B%=INT(A%/4)	B%=A%\4

4. Use built-in functions. These execute faster than when written in BASIC.

5. Use remarks sparingly. It takes some time for GWBASIC to identify a remark. If you use the single quote to place remarks at the end of the line rather than using a separate statement, not only will performance be improved but you will also be saving storage. For example:

```
10 FOR I=1 TO 10
15 'initialize A
20 A(I)=I*I
30 NEXT I
```

may also be written

```
10 FOR I=1 TO 10
20 A(I)=I*I 'initialize A
30 NEXT I
```

6. Since GWBASIC searches only once in a branching situation (after which the branch is direct), placing frequently used subroutines at the beginning of the program will not make a program run faster.

Logic Control

1. Use the capabilities of the IF statement. Do this by using the AND, OR, and ELSE clauses and save yourself additional coding.

   For example

```
200 IF A=B THEN GOTO 210
205 GOTO 215
210 IF C=D THEN 225
215 Z=B
220 GOTO 230
225 Z=12
230 ...
```

   is more efficiently written

```
200 IF A=B AND C=D THEN Z=12 ELSE Z=B
```

2. Order IF statements to test the most frequently occurring condition first. In this way you will avoid having to make extra tests. If, for example, you have a data entry file for customer orders consisting of different record types and many individual transactions, record types might be coded as follows:

Code	Record Type
A	Header
B	Customer name and address
C	Transaction
C	Transaction
.	.
.	.
C	Transaction
D	Trailer

Instead of coding this information

```
100 IF TYPE$="A" THEN 1000
110 IF TYPE$="B" THEN 2000
120 IF TYPE$="C" THEN 3000
130 IF TYPE$="D" THEN 4000
```

use

```
100 IF TYPE$="C" THEN 3000
110 IF TYPE$="A" THEN 1000
120 IF TYPE$="B" THEN 2000
130 IF TYPE$="D" THEN 4000
```

In this way with 100 groups and 10 transactions per group 1800 fewer IF statements will be executed.

Loops

1. Use integer counters on FOR...NEXT loops when possible. Performance will be faster than when single- and double-precision arithmetic is used.

2. Omit the variable on the NEXT statement where possible. If the variable is included, GWBASIC requires some time to check whether it is correct. Note that it may be necessary to include the variable on the NEXT statement if you are branching out of nested loops. See FOR and NEXT Statements in Chapter 4.

3. Use FOR...NEXT loops instead of IF, GOTO combinations. For example

```
200 I=1
210 ...
     .
     .
290 I=I+1
300 IF I<=10 THEN 210
```

is more easily written

```
200 FOR I=1 TO 10
  .
  .
  .
  .
300 NEXT I
```

4. Remove unnecessary code from loops. Statements that do not affect the loop and nonexecutable statements such as REM and DATA should be pruned. In the following example

```
10 FOR X=1 TO 100
20 A=B+1
30 IF D(X)>A THEN D(X)=A
40 NEXT X
```

since the loop never changes the value of A, it is not necessary to calculate this value each time. The code may be rewritten

```
10 A=B+1
20 FOR X=1 TO 100
30 IF D(X)>A THEN D(X)=A
40 NEXT X
```

Appendix H
RECOMMENDED READING

This appendix lists books that are useful introductions to BASIC, as well as other relevant documentation for your system.

INTRODUCTIONS TO BASIC

If you are new to BASIC, or to programming in general, the following books may be helpful.

BASIC and the Personal Computer, by Thomas A. Dwyer and Margot Critchfield. Reading, Mass.: Addison-Wesley Publishing Co., 1978.

BASIC, by Robert L. Albrecht, LeRoy Finkel and Jerry Brown. New York: Wiley Interscience, 2nd ed., 1978.

Are You Computer Literate?, by Karen Billings and David Moursund. Beaverton, Oregon: Dilithium Press, 1979.

Basic BASIC, by James Coan. Rochelle Park, N.J.: Hayden Book Co., 1978.

OTHER MANUALS

Other documentation referred to in the text is listed below.

DOS 2.0

PC-400/PPC-400 Technical Reference Manual.

Technical Reference Manual, 325-line systems.

MEGA PC Technical Reference Manual.

Appendix I
SUMMARY OF GWBASIC LANGUAGE

This is a summary of GWBASIC commands, statements, functions and variables.

v = ABS(**x**)

Returns the absolute value of **x**.

v = ASC(**a$**)

Returns the ASCII code for the first character of a string.

v = ATN(**x**)

Returns the arctangent of **x** in radians.

AUTO[**line number[,increment]**]

Generates a line number automatically every time <Return> is pressed.

BEEP

Sounds the speaker.

BLOAD **filespec[,offset]**

Loads a file in binary format into memory.

BSAVE **filespec,offset,length**

Transfers the contents of the specified area of memory to an output device, saving the data in binary format.

CALL **numvar[(variable[,variable]...)]**

Calls an assembly language program.

CALLS **numvar[(variable[,variable]...)]**

Calls an assembly language program, passing segmented addresses of all arguments.

v = CDBL(**x**)

Converts **x** to a double-precision number.

CHAIN [MERGE]**filespec[,[line][,ALL][,DELETE range]]**

Calls a program and passes variables to it.

CHDIR **pathname**

Changes the current directory.

v = CHR$(**a**)

Returns the character for a given ASCII code.

v = CINT(**x**)

Converts **x** to an integer.

CIRCLE [STEP] **(x,y),radius[,color[,start,end[,aspect]]]**

Draws a circle, arc or ellipse with the specified center and radius.

CLEAR **[,[expression1][,expression2]]**

Sets numeric variables to zero, string variables to null, and closes all open files. Options set end of memory and GWBASIC stack space.

CLOSE **[[#]filenum[,[#]filenum...]]**

Closes a disk file or device.

CLS

Erases contents of entire current screen and homes the cursor.

COLOR **[background][,palette]**

Sets background color for graphics mode and selects a color palette for the foreground.

COLOR **[foreground][,[background][,border]]**

Sets the foreground (i.e. character) color, background color, and border color respectively for text mode.

COM(**n**) ON
COM(**n**) OFF
COM(**n**) STOP

Enables or disables event trapping of communications activity on the specified channel.

COMMON **list of variables**

Passes variables to a chained program.

CONT

Continues program execution after a break.

v = COS(x)

Returns the cosine function of an angle.

v = CSNG(x)

Converts **x** to a single-precision number.

v = CSRLIN

Returns the current line position of the cursor.

v = CVI(2-byte string)
v = CVS(4-byte string)
v = CVD(8-byte string)

Converts string variables to numeric variables.

DATA list of constants

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

DATE$=string expression

Sets the date.

v$ = DATE$

Returns the system date.

DEF FNname[(parameter list)]=function definition

Defines and names a function written by the user.

DEF SEG [=address]

Defines address of current segment of memory.

DEF**type range(s) of letters**

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

DEF USR**[digit]=integer expression**

Specifies the starting address of an assembly language subroutine.

DELETE **[line number][-][line number]**

Deletes program lines.

DIM **list of subscripted variables**

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

DRAW **string expression**

Draws a line.

EDIT **line number**

Enters edit mode at the specified line.

END

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

ENVIRON **param** [=] **string**

Adds or modifies parmeter in BASIC environment table.

v$ = ENVIRON$ (param)
v$ = ENVIRON$ (n)

Returns a parameter value from the BASIC environment table.

v = EOF(file number)

Tests for the end-of-file condition.

ERASE **arrayname[,arrayname]...**

Deletes arrays from memory.

v = ERDEV
**v$** = ERDEV$

Return the last device error code issued (ERDEV) and the name of the device causing the error (ERDEV$).

ERR
ERL

Return error code (ERR) and line number (ERL) where error occurred.

ERROR **n**

Simulates occurrence of a GWBASIC error, or allows user definition of error codes.

v = EXP(x)

Returns the exponential function of **e** (base of natural logarithms).

FIELD[#]**file number,field width** AS **string variable**..

Allocates space for variables in a random file buffer.

FILES ["filespec"]

Displays names of files on the specified disk.

v = FIX(x)

Truncates x to an integer.

FOR variable=x TO y [STEP z]
.
.
NEXT [variable][,variable]...

Defines parameters for a loop.

v = FRE(x)
v = FRE(x$)

Returns number of bytes in memory not being used by GWBASIC, optionally tidying up the memory.

GET [#]file number [,record number]

Reads a record from a random file.

GET (x1,y1)-(x2,y2),arrayname

Transfers a graphic image from the screen to an array.

GOSUB line
.
.
RETURN [line]

Branches to and returns from a subroutine.

GOTO line

Branches unconditionally to specified line number.

GWBASIC [<stdin] [>[>stdout]] [filespec] [/C:combuffer] [/D] [/F:files] [/I][/M:[max workspace][,max block size]] [/S:bsize]

Runs GWBASIC from DOS.

v = HEX$(x)

Returns a string that represents the hexadecimal value of the decimal argument.

IF expression [,]THEN clause [ELSE clause] [:statement...]

IF expression [,]GOTO line [[,] ELSE clause]

Performs a branch or executes one or more statements if a specified condition is satisfied.

INKEY$

Reads an input character.

v = INP(n)

Returns the byte read from port address n.

INPUT[;]["prompt";]variable[,variable]...

Reads input during program execution.

v = INPUT$(n[,[#]filenum])

Reads specified number of characters from input device or file.

INPUT#filenum,variable[,variable]...

Reads data items from device or file and assigns them to program variables.

v = INSTR([n,]a$,b$)

Returns position of character string within another string.

v = INT(x)

Returns the largest integer less than or equal to x.

IOCTL [#]filenum, string

Transmits command string to a device driver.

v$ = IOCTL$ ([#]filenum)

Returns command string from a device driver.

KEY n,x$
KEY n, CHR$(mask) + CHR$(scan)
KEY LIST
KEY ON
KEY OFF

Sets or displays values of function keys, or sets values for a user-defined key or key sequence.

KEY(n) ON
KEY(n) OFF
KEY(n) STOP

Enables and disables trapping a specified key.

KILL filespec

Deletes a file from a disk.

LCOPY n

Prints the current screen contents.

v$** = LEFT$(x$,n**)

Returns a string comprising the leftmost **n** characters of **x$**.

v = LEN(**x$**)

Returns the number of characters in **x$**.

[LET] **variable=expression**

Assigns the value of an expression to a variable.

LINE[[STEP](**x1,y1**)]-[STEP](**x2,y2**)[,[**color**][,B[F]]] [,**style**]

Draws a line or a box on the screen.

LINE INPUT[;]["**prompt**";]**stringvar**

Inputs an entire line, ignoring delimiters, to a string variable.

LINE INPUT #**filenum,stringvar**

Reads an entire line, ignoring delimiters, to a string variable.

LIST[**line1**]
LIST [**line1**][-[**line2**]][,"**device**"]

Lists all or part of the program currently in memory.

LLIST [**line1**][-[**line2**]]

Lists all or part of the program currently in memory on the printer.

LOAD **filespec**[,R]

Loads a program from a specified device into memory, and optionally runs it.

v = LOC(**filenum**)

Returns the current position in a file.

LOCATE **[y][,[x][,[cursor][,[start][,stop]]]]**

Moves cursor to specified position on the screen, optionally turning the cursor on or off and defining its size.

v = LOF (**filenum**)

Returns the length of a file in bytes.

v = LOG(**x**)

Returns the natural logarithm of **x**.

v = LPOS(**n**)

Returns the current position of the print head in the printer's buffer.

LPRINT **[list of expressions][;]**
LPRINT USING **string exp;list of expressions[;]**

Prints data on the printer.

LSET **stringvar = x$**
RSET **stringvar = x$**

Moves data from memory to a random file buffer ready for a PUT statement, or left- or right-justifies a string in a given field.

MERGE **filespec**

Merges a program from a specified ASCII file into the program currently in memory.

v$ = MID$(x$,n[,m])

Returns a string of length **m** characters from **x$**, beginning with the **n**th character.

MID$(string-exp1,n[,m])=string-exp2

Replaces a portion of one string with another string.

MKDIR **pathname**

Creates a new directory

v$ = MKI$ (integer expression)
v$ = MKS$ (single-precision expression)
v$ = MKD$ (double-precision expression)

Converts numeric values to string values.

NAME **old-filespec** AS **new-filespec**

Renames a disk file.

NEW

Deletes the program currently in memory and clears all variables.

v$ = OCT$(n)

Returns the octal value of a decimal.

ON COM(**n**) GOSUB **line-number**

Specifies the first line number of a subroutine to be performed when activity occurs on a communications channel.

ON ERROR GOTO **line**

Enables errors to be trapped and specifies the first line of the error-handling subroutine.

ON **n** GOTO **line[,line]...**
ON **n** GOSUB **line[,line]...**

Branches to one of several specified line numbers, depending on a specified value.

ON KEY(**n**) GOSUB **line number**

Specifies first line number of a subroutine to be performed when specified function key, cursor direction key or user-defined key is pressed.

ON PEN GOSUB **line number**

Specifies first line number of subroutine to be performed when light pen is activated.

ON PLAY(**n**) GOSUB **line number**

Branches to a specified subroutine when the music queue contains fewer than **n** notes.

ON STRIG(**n**) GOSUB **line number**

Specifies the first line number of a subroutine to be performed when the joystick trigger is pressed.

ON TIMER(**n**) GOSUB **line number**

Branches to a subroutine when a specified time interval has elapsed.

OPEN **mode1,[#]file number,filespec[,record length]**

OPEN **filespec[FOR mode2]** AS**[#]file number[**LEN=**record length]**

Opens a data file or a device for input/output.

OPEN"COMn:**[speed][,[parity][,[data][,[stop]**[,RS][,CS**[n]]**[,DS**[n]]**[,CD**[n]] **[**,BIN**][**,ASC**][**,LF**]]]]"[FORmode]** AS**[#]filenum[**LEN=**record length]**

Opens a communications channel for input/output.

OPTION BASE n

Declares the minimum value for array subscripts.

OUT **i,j**

Sends a byte to a machine output port.

PAINT **(x,y)**[,**paint** [,**border**][,**background**]]

Fills a graphics area with the color or pattern specified.

v = PEEK(**n**)

Returns the byte read from memory location **n**.

x = PEN(**n**)
PEN ON
PEN OFF
PEN STOP

Reads the light pen and enables, disables, or stops trapping the pen.

PLAY **string**

Plays music from the specified string.

v = PLAY(n)

Returns the number of notes currently in the background music queue.

PLAY ON
PLAY OFF
PLAY STOP

Enables, disables or suspends music event trapping.

v = PMAP (**coord, function**)

Maps world coordinates created by the WINDOW statement to physical locations, or maps physical locations to world coordinates.

v = POINT (**n**)
v = POINT (**x-coordinate,y-coordinate**)

Reads the color value of a point on the screen, or returns the current graphics cursor coordinates.

POKE **i,j**

Writes a byte into a memory location.

v = POS(**n**)

Returns the current column position of the cursor.

PRESET [STEP](**x-coordinate,y-coordinate**)[**,color**]

Draws a specified point on the screen (default color is background color).

PRINT [**list of expressions**[{,| ;}]]

Outputs data on the screen.

PRINT USING **v$;list of expressions[;]**

Prints strings or numbers using a specified format.

PRINT **#filenum,**[USING **v$;]list of expressions**

Writes data sequentially to a file.

PSET[STEP]**(x-coordinate,y-coordinate)[,color]**

Draws a specified point on the screen (default color is foreground color).

PUT **[#]filenum[,number]**

Writes a record to a random access file.

PUT **(x,y),array[,action]**

Writes a graphics image from an array to a specified area of the screen.

RANDOMIZE **[expression]**
RANDOMIZE TIMER

Reseeds the random number generator.

READ **variable[,variable]...**

Reads values from a DATA statement and assigns them to variables.

REM **remark**

Allows explanatory remarks to be inserted in a program.

RENUM **[newnum][,[oldnum][,increment]**

Renumbers program lines.

RESET

Closes all files on all drives.

RESTORE [**line number**]

Allows DATA statements to be reread from a specified line.

RESUME
RESUME 0
RESUME NEXT
RESUME **line number**

Continues program execution after an error recovery procedure has been performed.

RETURN **[line number]**

Causes GWBASIC to return to the statement following the most recent GOSUB statement.

v$** = RIGHT$(a$,x**)

Returns the rightmost **x** characters of string **a$**.

RMDIR **pathname**

Removes an existing directory.

v = RND[**(x)**]

Returns a random number between 0 and 1.

RUN **[line]**
RUN **filespec**[,R]

Executes the program currently in memory, or loads a program into memory and runs it.

SAVE **filespec[,{A | P}]**

Stores a program file on disk.

v = SCREEN(**row,col[,z]**)

Returns the color or the ASCII code of the character at the specified row (line) and column of the screen.

SCREEN **[mode][,[burst][,[apage][,vpage]]]**

Sets screen attributes for use in subsequent statements.

v = SGN(**x**)

Returns the sign of **x**.

SHELL **[command-string]**

Executes a DOS command or runs a program and returns to GWBASIC.

v = SIN(**x**)

Returns the sine of **x**.

SOUND **freq,duration**

Generates sound through the speaker.

v$** = SPACE$(x**)

Returns a string of **x** spaces.

PRINT SPC(**x**)

Skips **x** spaces in a PRINT statement.

v = SQR(x)

Returns the square root of **x**.

v = STICK(n)

Accepts input from the joystick in the form of x- and y-coordinates.

STOP

Terminates program execution and returns to command level.

v$ = STR$(x)

Returns a string representation of the value of **x**.

v = STRIG (n)
STRIG ON
STRIG OFF
STRIG STOP

Returns the status of the specified joystick trigger or enables, disables, or stops trapping of the joystick.

v$ = STRING$(n,m)
v$ = STRING$(n,a$)

Returns a string of length **n** whose characters all have ASCII code **m** or the first character of **a$**.

SWAP variable1,variable2

Exchanges the values of two variables.

SYSTEM

Terminates GWBASIC and returns control to DOS.

[PRINT] TAB(**x**)

Moves cursor or print head to position **x**.

v = TAN(**x**)

Returns the tangent of **x**.

TIME$ = **x$**

Sets the time.

**v$** = TIME$

Retrieves the current time.

v = TIMER

Returns the number of seconds elapsed since midnight or the last system reset.

TIMER ON
TIMER OFF
TIMER STOP

Enables, disables or suspends timer event trapping.

TRON
TROFF

Traces the execution of program statements.

v = USR**[n](arg)**

Calls an assembly language subroutine.

v = VAL(**x$**)

Returns the numerical value of string **x$**.

v = VARPTR(**variable**)
v = VARPTR(**#filenum**)

Returns the starting address in memory of the variable or BASIC file control block.

v$** = VARPTR$(variable**)

Returns a string that defines the type of variable and its address in memory.

VIEW[[SCREEN][(**x1,y1**)-(**x2,y2**)[,[**color**][,[**border**]]]]]

Defines a screen viewport for graphics display.

VIEW PRINT [**top-line** TO **bottom-line**]

Defines a screen area for text display.

WAIT **port,i**[**,j**]

Suspends program execution while monitoring the status of a machine input port.

WHILE **expression**

.

(loop statements)

.

WEND

Executes a series of statements in a loop as long as a given condition is true.

WIDTH [LPRINT]**size**
WIDTH **#filenum,size**
WIDTH **device,size**

Sets the line width for the screen or printer in number of characters.

WINDOW [[SCREEN] **(x1,y1)-(x2,y2)**]

Redefines the dimensions of the current viewport for subsequent graphics statements.

WRITE **[list of expressions]**

Outputs data on the screen.

WRITE **#filenum,list of expressions**

Writes data to a sequential file.

Appendix J
ERROR MESSAGES

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 is encountered that contains an incorrect sequence of characters (such as an unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.). GWBASIC automatically enters edit mode at the line that caused the error.
3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous unmatched GOSUB statement.
4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal function call An out-of-range parameter is passed to a math or string function. A function call error may also occur as the result of: 1. A negative or unreasonably large subscript. 2. A negative or zero argument with LOG. 3. A negative argument to SQR.

Number	Message
	4. A negative mantissa with a non-integer exponent. 5. A call to a USR function for which the starting address has not yet been given. 6. An improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.
6	Overflow The result of a calculation is too large to be represented in GWBASIC number format. If underflow occurs, the result is zero, and execution continues without an error.
7	Out of memory A program is either too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.
8	Undefined line A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.
9	Subscript out of range An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.
10	Duplicate definition Two DIM statements are given for the same array; or a DIM statement is given for an array after the default dimension of 10 has been established for that array.
11	Division by zero A division by zero is encountered in an expression; or, the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of

Number	Message
	the numerator is supplied as the result of the division,or positive machine infinity is supplied as the result of the involution, and execution continues.
12	Illegal direct A statement that is illegal in direct mode is entered as a direct mode command.
13	Type mismatch A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.
14	Out of string space String variables have caused GWBASIC to exceed the amount of free memory remaining. GWBASIC allocates string space dynamically until it runs out of memory.
15	String too long An attempt is made 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 should be broken into smaller expressions.
17	Can't continue An attempt is made to continue a program that: 1. Has halted due to an error. 2. Has been modified during a break in execution. 3. Does not exist.

Number	Message
18	Undefined user function A USR function is called before the function definition (DEF statement) is given.
19	No RESUME An error-handling routine is entered but contains no RESUME statement.
20	RESUME without error A RESUME statement is encountered before an error-handling routine is entered.
21	Unprintable error An error message is not available for the existing error condition.
22	Missing operand An expression contains an operator with no operand following it.
23	Line buffer overflow An attempt has been made to input a line that has too many characters.
24	Device time-out GWBASIC did not receive information from an I/O device within a predetermined amount of time.
25	Device fault An incorrect device designation has been entered.
26	FOR without NEXT A FOR statement was encountered without a matching NEXT.

Number	Message
27	Out of paper The printer is either out of paper or not switched on.
29	WHILE without WEND A WHILE statement does not have a matching WEND.
30	WEND without WHILE A WEND statement was encountered without a matching WHILE.
50	FIELD overflow A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
51	Internal error An internal malfunction has occurred in GWBASIC. Report to manufacturer the conditions under which the message appeared.
52	Bad file number A statement or command references a file with a number not yet open or out of the range of file numbers specified at initialization.
53	File not found A LOAD, KILL, or OPEN statement references a file that does not exist on the current diskette.
54	Bad file mode An attempt is made to use PUT, GET, or LOF with a sequential file, to load a random file, or to execute an OPEN statement with a file mode other than I, O, or R.

Number	Message
55	File already open. An OPEN statement for sequential output is issued for a file that is already open; or a KILL statement is given for a file that is open.
57	Device I/O error An I/O error occurred on a device. This is a fatal error, i.e. the operating system cannot recover from it.
58	File already exists The file name specified in a NAME statement is identical to one already in use on the disk.
61	Disk full All disk storage space is in use.
62	Input past end An INPUT statement is executed after all the data in the file has been input, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.
63	Bad record number In a PUT or GET statement, the record number is either greater than the maximum allowed (16,777,215) or equal to zero.
64	Bad file name An illegal form has been used for the file name with a LOAD, SAVE, KILL, or OPEN statement (e.g., a file name with too many characters).

Number	Message
66	Direct statement in file

A direct statement is encountered while loading an ASCII file. The LOAD is terminated.

67	Too many files

An attempt is made to create a new file (using SAVE or OPEN) when all available directory entries are full.

68	Device unavailable

An attempt was made to open a file to a nonexistent device. It may be that hardware does not exist to support the device, such as LPT3:, or that it was disabled. This occurs if an OPEN "COM1:... statement is executed but the user disabled RS232 support via the /C:0 switch directive on the command line.

69	Communications buffer overflow

Occurs when a communications input statement is executed but the input queue was already full. Use an ON ERROR GOTO statement to retry the input when this condition occurs. Subsequent inputs will 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 via the /C: switch.

2. Implement a "handshaking" protocol, such as XON/XOFF, with the host/satellite to turn transmit off long enough to catch up. (The TTY programming example in Section 6 demonstrates this option.)

3. Use a lower baud rate for transmit and receive.

70	Disk write protect

An attempt has been made to write to a diskette that is write-protected or that cannot be written to. Use an ON

Number	Message
	ON ERROR GOTO statement to detect this situation and request user action.
71	Disk not ready Occurs when the diskette drive door is open, or a diskette is not in the drive or has not been properly inserted. Use an ON ERROR GOTO statement to recover.
72	Disk media error A hardware or disk problem occurred while the disk was being written to or read from. For example, the disk drive may be damaged or the disk drive may not be working properly.
74	Rename across disks An attempt was made to rename a file with a new drive designation. This is not allowed.
75	Path/file access error During an OPEN, MKDIR, CHDIR or RMDIR operation, DOS was unable to make a correct path-to-filename connection. The operation is not completed.
76	Path not found During an OPEN, MKDIR, CHDIR or RMDIR operation, DOS was unable to find the path specified. The operation is not completed.
**	You cannot run BASIC as a child of BASIC You have attempted to SHELL from BASIC to BASIC. This is not allowed, and control returns to the parent BASIC.

** Can't continue after SHELL

Upon returning from a child process, BASIC discovers that there is not enough memory to continue. BASIC closes any open files and exits to DOS.

INDEX

INDEX