

TERMS AND CONDITIONS OF SALE AND LICENSE OF TANDY COMPUTER EQUIPMENT AND
SOFTWARE PURCHASED FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL
STORES AND RADIO SHACK FRANCHISEES OR DEALERS AT THEIR AUTHORIZED LOCATIONS

USA LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

- A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.
- B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. LIMITED WARRANTIES AND CONDITIONS OF SALE

- A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purchases of Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores, and Radio Shack franchisees and dealers at their authorized locations.** The warranty is void if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or a participating Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.
- B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or Radio Shack dealer along with the sales document.
- C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.
- D. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK MAKES NO EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.**
- E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

- A. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE."**
NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.
- B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.
- C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.
- D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the TANDY Software on **one** computer, subject to the following provisions:

- A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.
- B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.
- C. CUSTOMER may use Software on a multiuser or network system only if either, the Software is expressly labeled to be for use on a multiuser or network system, or one copy of this software is purchased for each node or terminal on which Software is to be used simultaneously.
- D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.
- E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.
- F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.
- G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

- A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby Radio Shack sells or conveys such Equipment to a third party for lease to CUSTOMER.
- B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by Radio Shack.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

Tandy
GW-BASIC

GW-BASIC[®] Software:
©1983, 1984, 1985, 1987 Microsoft[®] Corporation.
Licensed to Tandy Corporation.
All Rights Reserved.

All portions of this software are copyrighted and are the proprietary and trade secret information of Tandy Corporation and/or its licensor. Use, reproduction, or publication of any portion of this material without prior written authorization by Tandy Corporation is strictly prohibited.

Tandy GW-BASIC User's Guide Manual
Copyright 1987 Tandy Corporation
All Rights Reserved.

Tandy GW-BASIC User's Reference Manual
Copyright 1987 Tandy Corporation
All Rights Reserved.

Reproduction or use of any portion of these manuals without express written permission from Tandy Corporation and/or its licensor is prohibited. While reasonable efforts have been made in the preparation of these manuals to assure their accuracy, Tandy Corporation assumes no liability resulting from any errors in or omissions from these manuals, or from the information contained herein.

Tandy is a registered trademark of Tandy Corporation.
Microsoft, MS-DOS, and GW are registered trademarks of Microsoft Corporation.
IBM is a registered trademark of International Business Machines Corporation.

Tandy
GW-BASIC
User's Guide

CONTENTS

1	Welcome to GW-BASIC	1
	Systems Requirements	1
	Preliminaries	1
	Notational Conventions	2
	Organization of This Manual	2
	Bibliography	3
2	Getting Started With GW-BASIC	5
	Loading GW-BASIC	5
	Loading BASIC with BASICA	6
	Modes of Operation	7
	Direct Mode	7
	Indirect Mode	7
	The GW-BASIC Command Line Format	7
	Redirection of Standard Input and Output	10
	GW-BASIC Statements, Functions, Commands, and Variables	11
	Keywords	11
	Commands	12
	Statements	12
	Functions	12
	Numeric Functions	12
	String Functions	13
	Variables	13
	Line Format	13
	Returning to MS-DOS	15
3	Reviewing and Practicing GW-BASIC	17
	Example for the Direct Mode	17
	Examples for the Indirect Mode	18
	Function Keys	19
	Editing Lines	20
	Saving Your Program File	22

4	The GW-BASIC Screen Editor	23
	Editing Lines in New Files	23
	Editing Lines in Saved Files	23
	Editing the Information in a Program Line	24
	Special Keys	25
	Function Keys	29
5	Creating and Using Files	31
	Program File Commands	31
	Data Files	32
	Creating a Sequential File	32
	Accessing a Sequential File	35
	Adding Data to a Sequential File	36
	Random Access Files	37
	Accessing a Random Access File	39
6	Constants, Variables, Expressions and Operators	45
	Constants	45
	Single- and Double-Precision Form for Numeric Constants	47
	Variables	47
	Variable Names and Declarations	48
	Array Variables	49
	Memory Space Requirements for Variable Storage	50
	Type Conversion	51
	Expressions and Operators	52
	Arithmetic Operators	53
	Integer Division and Modulus Arithmetic	54
	Overflow and Division by Zero	54
	Relational Operators	55
	Logical Operators	56
	Functional Operators	58
	String Operators	59

Appendices

A	Error Codes and Messages	61
B	ASCII Character Codes	69
C	Assembly Language (Machine Code) Subroutines	77
D	Converting BASIC PROGRAMS to GW-BASIC	89
E	Communications	91
F	Hexadecimal Equivalents	101
G	Key Scan Codes	105
H	Characters Recognized by GW-BASIC	107
I	Glossary	109

Welcome to GW-BASIC

Microsoft® GW-BASIC® is a simple, easy-to-learn, easy-to-use computer programming language with English-like statements and mathematical notations. With GW-BASIC you can write both simple and complex programs to run on your computer. You can also modify existing software that is written in GW-BASIC.

This guide is designed to help you use the GW-BASIC Interpreter with the MS-DOS® operating system. The Bibliography in this chapter lists resources that can help you learn how to program.

System Requirements

This version of GW-BASIC requires MS-DOS version 3.2 or later.

Preliminaries

Your GW-BASIC files are on the MS-DOS diskette, which is located at the back of the *Tandy 3000/4000 MS-DOS User's Reference*. Be sure to make a working copy of the diskette before you proceed.

Note: Be sure you are familiar with MS-DOS before you begin using this manual. For more information on MS-DOS, refer to the *Tandy 3000/4000 MS-DOS Handbook* and the *Tandy 3000/4000 MS-DOS Reference Manual*.

Notational Conventions

Throughout this manual, the following conventions are used to distinguish elements of text:

bold	Used for commands, options, switches, and literal portions of syntax that must appear exactly as shown
<i>italic</i>	Used for variables, new terms, and manual names
sans-serif type	Used for sample command lines, program code and examples, and sample sessions
BOX TEXT	Used for keys and key sequences

Brackets surround optional command line elements.

Organization of This Manual

The *GW-BASIC User's Guide* is divided into six chapters and nine appendices, including a glossary:

Chapter 1, "Welcome to GW-BASIC," describes this manual.

Chapter 2, "Getting Started With GW-BASIC," is an elementary guide on how to begin programming.

Chapter 3, "Reviewing and Practicing GW-BASIC," lets you use the principles of GW-BASIC explained in Chapter 2.

Chapter 4, "The GW-BASIC Screen Editor," discusses editing commands you can use when entering or modifying a GW-BASIC program. It also explains the unique properties of the ten redefinable function keys and of other keys and keystroke combinations.

Chapter 5, "Creating and Using Files," tells you how to create files and use the diskette input/output (I/O) procedures.

Chapter 6, "Constants, Variables, Expressions, and Operators," defines the elements of GW-BASIC and describes how you will use them.

Appendix A, "Error Codes and Messages," is a summary of all the error codes and error messages you might encounter while using GW-BASIC.

Appendix B, "ASCII Character Codes," lists the ASCII character codes recognized by GW-BASIC.

Appendix C, "Assembly Language (Machine Code) Subroutines," shows how to include assembly language subroutines with GW-BASIC.

Appendix D, "Converting BASIC Programs to GW-BASIC," provides pointers on converting programs written in BASIC to GW-BASIC.

Appendix E, "Communications," describes the GW-BASIC statements required to support RS-232 asynchronous communications with other computers and peripheral devices.

Appendix F, "Hexadecimal Equivalents," lists decimal and binary equivalents to hexadecimal values.

Appendix G, "Key Scan Codes," lists and illustrates the key scan code values used in GW-BASIC.

Appendix H, "Characters Recognized by GW-BASIC," describes the GW-BASIC character set.

Appendix I, "Glossary," defines words and phrases commonly used in GW-BASIC and data processing.

Bibliography

This manual is a guide to the use of the GW-BASIC Interpreter. It makes no attempt to teach the BASIC programming language. You might find the following books useful for learning BASIC programming:

Albrecht, Robert L., LeRoy Finkel, and Jerry Brown. *BASIC*. 2d ed. New York: Wiley Interscience, 1978.

Coan, James. *Basic BASIC*. Rochelle Park, N.J.: Hayden Book Company, 1978.

Dwyer, Thomas A. and Margot Critchfield. *BASIC and the Personal Computer*. Reading, Mass.: Addison-Wesley Publishing Co., 1978.

Ettlin, Walter A. and Gregory Solberg. *The MBASIC Handbook*. Berkeley, Calif.: Osborne/McGraw Hill, 1983.

Knecht, Ken. *Microsoft BASIC*. Portland, Oreg.: Dilithium Press, 1982.

Getting Started With GW-BASIC

This chapter describes how to load GW-BASIC into your system. It also explains the two different types of operation modes, the line formats, and the various elements of GW-BASIC.

Loading GW-BASIC

To use the GW-BASIC language, load it into the memory of your computer from your working copy of the MS-DOS diskette. Use the following procedure:

1. Turn on your computer.
2. Insert your working copy of the MS-DOS diskette into Drive A of your computer, and press **ENTER**.
3. From the system prompt (such as A> or C>) type the following command and press **ENTER** :

basic

Once you enter GW-BASIC, the GW-BASIC prompt, **Ok**, replaces the MS-DOS prompt, **A>**.

On the screen, the line **XXXXX Bytes Free** indicates the number of bytes available for use in memory while using GW-BASIC.

The function key (**F1** — **F10**) assignments appear on the bottom line of the screen. You can use these function keys to eliminate keystrokes and save time. Chapter 4, "The GW-BASIC Screen Editor," contains detailed information on function keys.

Loading BASIC with BASICA

For compatibility, some computers require that you use the BASICA program to load GW-BASIC. To use BASICA, type the following and press **ENTER** :

BASICA

Your computer first loads `Basica.com`, which in turn loads GW-BASIC.

BASICA provides the following advantages:

- BASICA loads BASIC at a different memory location. This lets you run some BASIC programs that use memory locations normally reserved for BASIC's operations.
- You can gain space on your program or system diskette because you can store the `Basic.exe` file on a separate diskette.

The only limitations imposed by BASICA are:

The `/i` option switch, discussed in "Options for Loading BASIC", is always on.

The communications buffer size is limited to 40K bytes if the system has one RS232 card or 20K bytes if it has two.

After you execute BASICA, it searches the current directory for the file `Basic.exe`. If it finds `Basic.exe`, `Basica.com` loads it and passes control to it.

If `Basica.com` does not find `Basic.exe`, it asks you to replace your program disk with a disk that contains the file. Place a diskette containing `Basic.exe` in any drive, and press **ENTER**. The program searches all drives, beginning with the current drive, until it finds `Basic.exe` or until you press **CTRL** **C**.

After finding `Basic.exe`, BASICA asks you to re-insert your program diskette if you removed it. Put the program diskette back into the drive and press **ENTER**. BASICA transfers control back to BASIC.

Modes of Operation

Once GW-BASIC is initialized (loaded), the Ok prompt appears on the screen. Ok means GW-BASIC is at *command level*; that is, it is ready to accept commands. At this point, you can use GW-BASIC in either of two modes: *direct mode* or *indirect mode*.

Direct Mode

In the direct mode, GW-BASIC statements and commands are executed as they are entered. Results of arithmetic and logical operations are displayed immediately and/or stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using GW-BASIC as a calculator for quick computations that do not require a complete program.

Indirect Mode

Use the indirect mode to enter programs. Program lines, which are always preceded by line numbers, are stored in memory. The program stored in memory is executed by entering the RUN command.

The GW-BASIC Command Line Format

The GW-BASIC command line lets you change the environment or the conditions that apply while using GW-BASIC.

Note: When you specify modifications to the operating environment of GW-BASIC, be sure to maintain the parameter sequence shown in the syntax statement.

GW-BASIC uses a command line similar to the following:

```
basic [ filename][ [<stdin>]] [>] > stdout[/f: number[/i]  
[/s: number[/c: number[/m:[number], number]]]/d]
```

filename is the name of a GW-BASIC program file. If this parameter is present, GW-BASIC proceeds as if a RUN command had been given. If no extension is provided for the filename, a default file extension of .BAS is assumed. The .BAS extension indicates that the file is a GW-BASIC file. A filename can contain a maximum of eight characters, plus a decimal and three extension characters.

<stdin redirects GW-BASIC's standard input to be read from the specified file. When used, it must appear before any switches.

This might be used when your program is using multiple files and you want to specify a particular input file.

>stdout redirects GW-BASIC's standard output to the specified file or device. When used, it must appear before any switches. Using *>>* before *stdout* causes output to be appended.

You can redirect GW-BASIC to read from standard input (keyboard) and write to standard output (screen) by providing the input and output filenames on the command line as follows:

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

An explanation of file redirection follows this discussion of the GW-BASIC command line.

Switches appear frequently in command lines. They designate a specified course of action for the command, as opposed to using the default for that setting. A switch parameter is preceded by a slash (/).

/f: number sets the maximum number of files that can be opened simultaneously during the execution of a GW-BASIC program. Each file requires 194 bytes for the File Control Block (FCB) plus 128 bytes for the data buffer. You can alter the data buffer size with the */s:* switch. If the */f:* switch is omitted, the maximum number of open files defaults to 3. This switch is ignored unless the */i* switch is also specified on the command line.

/i makes GW-BASIC statically allocate space required for file operations, based on the */s:* and */f:* switches.

/s: number sets the maximum record length allowed for use with files. The *record length* option in the OPEN statement cannot exceed this value. If you omit the */s:* switch, the record length defaults to 128 bytes. The maximum record size is 32767.

/c: number controls RS-232 communications. If RS-232 cards are present, */c:0* disables RS-232 support and any subsequent I/O attempts for each RS-232 card present. If you omit the */c:*, 256 bytes are allocated for the receive buffer and 128 bytes for the transmit buffer for each card present.

The */c:* switch has no effect when RS-232 cards are not present. The */c: number* switch allocates *number* bytes for the receive buffer and 128 bytes for the transmit buffer for each RS-232 card present.

/m: number[, number] sets the highest memory location (first *number*) and maximum block size (second *number*) used by GW-BASIC. GW-BASIC attempts to allocate 64K bytes of memory for the data and stack segments. If you use machine language subroutines with GW-BASIC programs, use the */m:* switch to set the highest location that GW-BASIC can use. The maximum block size is in multiples of 16. It is used to reserve space for user programs (assembly language subroutines) beyond GW-BASIC's workspace.

The default for the maximum block size is the highest memory location. The default for the highest memory location is 64K bytes unless the maximum block size is specified, in which case the default is the maximum block size (in multiples of 16).

/d allows certain functions to return double-precision results. When you specify the */d* switch, approximately 3000 bytes of additional code space are used. The functions affected are ATN, COS, EXP, LOG, SIN, SQR, and TAN.

Note: You can specify all switch numbers as decimal, octal (preceded by &O), or hexadecimal (preceded by &H).

The following examples are sample GW-BASIC command lines.

The following example uses 64K bytes of memory and three files, and loads and executes the program file Payroll.bas:

A > basic payroll

The next example uses 64K bytes of memory and six files, and loads and executes the program file Invent.bas:

A > basic invent /f:6

The following example disables RS-232 support and uses only the first 32K bytes of memory. The 32K bytes above that are reserved for user programs:

```
A> basic /c:0 /m:32768,4096
```

The next example uses four files and allows a maximum record length of 512 bytes:

```
A> basic /f:4 /s:512
```

The following example uses 64K bytes of memory and three files. It allocates 512 bytes to RS-232 receive buffers and 128 bytes to transmit buffers, and loads and executes the program file Tty.bas:

```
A> basic tty /c:512
```

For more information about RS-232 communications, see Appendix E.

Redirection of Standard Input and Output

When redirected, all INPUT, LINE INPUT, INPUT\$, and INKEY\$ statements are read from the specified input file instead of from the keyboard.

All PRINT statements write to the specified output file instead of to the screen.

Error messages go to standard output and to the screen.

File input from KYBD: is still read from the keyboard.

File output to SCRN: still outputs to the screen.

GW-BASIC continues to trap keys when the ON KEY *number* statement is used.

Pressing **CTRL** **BREAK** when output is redirected causes GW-BASIC to close any open files, issue the message Break in line *nnnn* to standard output, exit GW-BASIC, and return to MS-DOS.

When input is redirected, GW-BASIC continues to read from this source until it detects a CTRL-Z character. You can test this condition with the end-of-file (EOF) function. If the file is not terminated by a CTRL-Z

character, or if a GW-BASIC file input statement tries to read past the end of file, then any open files are closed, and GW-BASIC returns to MS-DOS.

For further information about these statements and other statements, functions, commands, and variables mentioned in this text, refer to the *GW-BASIC User's Reference*.

Some examples of redirection follow.

```
basic myprog > data.out
```

Data read by the INPUT and LINE INPUT statements continues to come from the keyboard. Data output by the PRINT statement goes into the Data.out file.

```
basic myprog < data.in
```

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

```
basic myprog < myinput.bat > myoutput.dat
```

Data read by the INPUT and LINE INPUT statements now comes from the file, Myinput.dat, and data output by the PRINT statements goes into Myoutput.dat.

```
basic myprog < \sales\john\trans.dat > > \sales\sales.dat
```

Data read by the INPUT and LINE INPUT statements now comes from the file \sales\john\trans.dat. Data output by the PRINT statement is appended to the file \sales\sales.dat.

GW-BASIC Statements, Functions, Commands, and Variables

A GW-BASIC program is made up of several elements: keywords, commands, statements, functions, and variables.

Keywords

GW-BASIC keywords, such as print, goto, and return have special significance for the GW-BASIC Interpreter. GW-BASIC interprets keywords as part of statements or commands.

Keywords are also called *reserved words*. They cannot be used as variable names, or the system will interpret them as commands. However, keywords can be embedded within variable names.

Keywords are stored in the system as *tokens* (1- or 2-byte characters) for the most efficient use of memory space.

Commands

Commands and statements are both executable instructions. The difference between commands and statements is that you generally execute commands in GW-BASIC's direct mode or the command level. They usually perform program maintenance such as editing, loading, or saving programs. When GW-BASIC is invoked and the GW-BASIC prompt, Ok, appears, the system assumes command level.

Statements

A statement, such as ON ERROR...GOTO, is a group of GW-BASIC keywords generally used in GW-BASIC program lines as part of a program. When the program is run, statements are executed when, and as, they appear.

Functions

The GW-BASIC Interpreter performs both numeric and string functions.

Numeric Functions

The GW-BASIC Interpreter can perform certain mathematical (arithmetic or algebraic) calculations. For example, it calculates the sine (sin), cosine (cos), or tangent (tan) of angle x .

Unless otherwise indicated, numeric functions return only integer and single-precision results.

String Functions

String functions operate on strings. For example, `TIMES` and `DATES` return the time and date known by the system. If you enter the current time and date during system startup, the correct time and date are given. (The internal clock in the computer keeps track.)

User-Defined Functions

Functions can be user-defined by means of the `DEF FN` statement. These functions can be either string or numeric.

Variables

Certain groups of alphanumeric characters are assigned values and are called *variables*. When variables are built into the GW-BASIC program, they provide information as they are executed.

For example, `ERR` defines the latest error that occurred in the program; `ERL` gives the location of that error. Either you or a program's content can define and/or redefine variables.

All GW-BASIC commands, statements, functions, and variables are individually described in the *GW-BASIC User's Reference*.

Line Format

Each element of GW-BASIC can make up sections of a program that are called *statements*. These statements are similar to English sentences. Statements are then put together in a logical manner to create programs. The *GW-BASIC User's Reference* describes all the statements available for use in GW-BASIC.

In a GW-BASIC program, lines have the following format:

nnnnn statement [statements]

nnnnn is a line number.

statement is a GW-BASIC statement.

A GW-BASIC program line always begins with a line number and must contain at least one character but no more than 255 characters. Line numbers indicate the order in which the program lines are stored in memory, and are also used as references when branching and editing. The program line ends when you press **ENTER** .

Depending on the logic of your program, there might be more than one statement on a line. If so, each must be separated by a colon (:).

Precede each line in a program with a line number. This number can be any whole integer, 0-65529. It is customary to use line numbers such as 10, 20, 30, and 40, in order to leave room for any additional lines that you might want to add later. Because the computer runs the statements in numerical order, additional lines needn't appear in consecutive order on the screen. For example, if you entered Line 35 after Line 60, the computer would still run Line 35 after Line 30 and before Line 40. This line numbering technique might save you from re-entering an entire program only to include one line that you have forgotten.

The width of your screen is 80 characters. If your statement exceeds this width, the cursor wraps to the next screen line automatically. Only when you press the **ENTER** key does the computer acknowledge the end of the line. Resist the temptation to press **ENTER** as you approach the edge of the screen (or beyond). The computer automatically wraps the line for you. You can also press **CTRL** **ENTER** , which causes the cursor to move to the beginning of the next screen line without actually entering the line. When you press **ENTER** , the entire logical line passes to GW-BASIC for storage in the program.

In GW-BASIC, any line of text that begins with a numeric character is considered a program line and is processed in one of three ways after you press **ENTER** :

- A new line is added to the program. This occurs if the line number is legal (within the range 0-65529) and if at least one alpha or special character follows the line number in the line.
- An existing line is modified. This occurs if the line number matches the line number of an existing line in the program. The existing line is replaced with the text of the newly entered line. This process is called *editing*.

Note: Reusing an existing line number causes all the information contained in the original line to be lost. Be careful when entering numbers in the indirect mode. You might erase some program lines accidentally.

- An existing line is deleted. This occurs if the line number matches the line number of an existing line and the entered line contains only a line number. If you attempt to delete a nonexistent line, the screen displays an Undefined line number error message.

Returning to MS-DOS

Before you return to MS-DOS, you must save the work you have entered under GW-BASIC; otherwise, the work will be lost.

To return to MS-DOS, type the following after the Ok prompt, and press

ENTER :

system

The system returns to MS-DOS, and the A> prompt appears on your screen.

Reviewing and Practicing GW-BASIC

The practice sessions in this chapter will help you review what you have learned. If you have not done so, this is a good time to turn on your computer and load the GW-BASIC Interpreter.

Example for the Direct Mode

You can use your computer in the direct mode to perform fundamental arithmetic operations. GW-BASIC recognizes the following symbols as arithmetic operators:

Operation	GW-BASIC Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/

To enter an arithmetic problem, respond to the Ok prompt with a question mark (?) followed by the statement of the problem you want to solve, and press **ENTER**. In GW-BASIC, you can use the question mark interchangeably with the keyword PRINT. The answer is then displayed.

Type the following, and press **ENTER**:

?2 + 2

GW-BASIC displays the answer on your screen:

4

Ok

To practice other arithmetic operations, replace the + sign with the desired operator.

The GW-BASIC language is not restricted to arithmetic functions. You can also enter complex algebraic and trigonometric functions. The formats for these functions are provided in Chapter 6, "Constants, Variables, Expressions and Operators."

Examples for the Indirect Mode

You can use the GW-BASIC language to perform functions other than simple algebraic calculations. You can create a program that performs a series of operations and then displays the answer. To begin programming, you create lines of instructions called statements. Remember that there can be more than one statement on a line and that each line is preceded by a number.

For example, to create the command `PRINT 2 + 3` as a statement, type:

```
10 print 2 + 3
```

When you press `ENTER`, the cursor shifts to the next line, but nothing else happens. To make the computer perform the calculation, type the following and press `ENTER`:

```
run
```

Your screen should display this answer:

```
5  
Ok
```

You have just written a program in GW-BASIC.

The computer reserves its calculation until specifically commanded to continue (with the `RUN` command). This enables you to enter more lines of instruction. When you type the `RUN` command, the computer performs the addition and displays the answer.

The following program has two lines of instructions. Enter it:

```
10 x = 3  
20 print 2 + x
```

Now, use the `RUN` command to make the computer calculate the answer.

Your screen should display:

```
5  
Ok
```

The two features that distinguish a program from a calculation are:

- the numbered lines
- the use of the RUN command

These features let the computer know that you have finished typing all the statements and that it can carry out the computation from beginning to end. The numbering of the lines first signals the computer that this is a program, not a calculation, and that it must not do the actual computation until you enter the RUN command.

In other words, calculations are performed in the direct mode. Programs are written in the indirect mode.

To display the entire program again, type the LIST command, and press **ENTER**:

```
list
```

Your screen should display:

```
10 X=3  
20 PRINT 2+X  
Ok
```

You'll notice a slight change in the program. The lowercase letters you entered have been converted into uppercase letters. The LIST command makes this change automatically.

Function Keys

Function keys are keys that have been assigned to frequently used commands. A guide to these ten keys and their assigned commands appears at the bottom of the GW-BASIC screen. To save time and keystrokes, you can press a function key instead of typing a command name.

For example, to list your program again, you needn't type the LIST command. Instead, you can use the function key assigned to it:

1. Press the **F1** key.
2. Press **ENTER**.

Your program should appear on the screen.

To run the program, simply press **F2**, which is assigned to the RUN command.

As you learn more commands, you learn to use keys **F3** - **F10**.

Chapter 4, "The GW-BASIC Screen Editor," contains more information about keys used in GW-BASIC.

Editing Lines

There are two basic ways to change lines. You can:





- Delete and replace them
- Alter them with the EDIT command


To delete a line, type the line number and press **ENTER**. For example, if you type 12 and press **ENTER**, Line 12 is deleted from your program.

To use the EDIT command to change a line, type the command followed by the number of the line you want to change. For example, type the following, and press **ENTER**:

```
edit 10
```

You can then use the following keys to perform editing:

Key	Function
	Moves the cursor to the previous line
	Moves the cursor to the next line
	Moves the cursor within the statement
	Moves the cursor within the statement
BACKSPACE	Deletes the character to the left of the cursor
DEL	Deletes the current character
INS	Lets you insert characters left of the cursor

For example, to modify statement (line) 10 to read $x = 4$, use the  key to move the cursor under the 3, and then type 4. The number 4 replaces the number 3 in the statement.

Now, press **ENTER** and then **F2**.

Your screen displays:

6
Ok

Saving Your Program File

Creating a program is like creating a data file. The program is a file that contains specific instructions, or statements, for the computer. In order to use the program again, you must save it, just as you would a data file.

To save a file in GW-BASIC, use the following procedure:

1. Press **F4** . The command word **SAVE** appears on your screen.
2. Type a name for the program, and press **ENTER** .

The file is saved under the name you specified.

To recall a saved file, use the following procedure:

1. Press **F3** . The command word **LOAD** appears on your screen.
2. Type the name of the file.
3. Press **ENTER** .

The file is loaded into memory, ready for you to list, edit, or run.

The GW-BASIC Screen Editor

You can edit GW-BASIC program lines either as you enter them or after you have saved them in a program file.

Editing Lines in New Files

If you make a mistake while typing a GW-BASIC line, you can use the `BACKSPACE`, `DEL`, or the `CTRL` `H` keys to erase the erroneous character. After you delete the character, you can continue to type on the line.

The `ESC` key lets you cancel a line that you are typing. In other words, if you have not yet pressed `ENTER` and you want to terminate the line you are entering, press `ESC`.

To delete the entire program currently residing in memory, enter the `NEW` command. `NEW` is usually used to clear memory prior to entering a new program.

Editing Lines in Saved Files

After you have entered your GW-BASIC program and saved it, you might discover that you need to make some changes. To make these modifications, use the `LIST` statement to display the program lines that are affected:

1. Reload the program.
2. Type the `LIST` command, or press `F1`.
3. Type the line number, or range of numbers, that you want to edit.

The lines appear on your screen.

Editing the Information in a Program Line

You can make changes to the information in a line by positioning the cursor where you want to make the change and doing one of the following:

- Typing over the characters that are already there.
- Deleting characters to the left of the cursor, using the **←** key.
- Deleting characters at the cursor position, using the **DEL** key on the number pad.
- Inserting characters at the cursor position by pressing the **INS** key on the number pad. This moves the characters following the cursor to the right, making room for the new information.
- Adding to or truncating characters at the end of the program line.

Always press **ENTER** on every line you modify. If you forget to press enter, the line you are modifying is not changed.

Modified lines are stored in the proper numerical sequence, even if you do not update them in numerical order.

Note: Changes you make to a program line are not actually recorded in GW-BASIC until you press **ENTER** with the cursor positioned somewhere on the edit line.

You do not have to move the cursor to the end of the line before pressing the **ENTER** key. The GW-BASIC Interpreter remembers where each line ends and transfers the whole line, even if you press **ENTER** while the cursor is in the middle or at the beginning of the line.

Type **CTRL** **END** or **CTRL** **E**, then **ENTER**, to truncate, or cut off, a line at the current cursor position.

If you originally saved your program to a program file, be sure to save the edited version of your program. If you do not, your modifications will not be recorded.

Special Keys

GW-BASIC recognizes the nine numeric keys located at the right side of your keyboard. It also recognizes the **BACKSPACE** key, the **ESC** key, and the **CTRL** key. The following keys and key sequences have special functions in GW-BASIC:

BACKSPACE or **CTRL** **H**

Deletes the last character typed, or deletes the character to the left of the cursor. All characters to the right of the cursor move left one position. Subsequent characters and lines within the current logical line move up as with the **DEL** key.

CTRL **BREAK** or **CTRL** **C**

Returns to the direct mode, without saving changes made to the current line. Also exits auto line-numbering mode.

CTRL **←** or **CTRL** **B**

Moves the cursor to the beginning of the previous word. The previous word is defined as the next letter or number to the left of the cursor that follows a blank or other special character.

CTRL **→** or **CTRL** **F**

Moves the cursor to the beginning of the next word. The previous word is defined as the next letter or number to the right of the cursor that follows a blank or other special character.

↓ or **CTRL** **-**

Moves the cursor down one line on the screen.

`←` or `CTRL` `[]`

Moves the cursor one position left. When the cursor advances beyond the left edge of the screen, it wraps to the end of the preceding line.

`→` or `CTRL` `[\]`

Moves the cursor one position right. When the cursor advances beyond the right edge of the screen, it wraps to the beginning of the following line.

`↑` or `CTRL` `[6]`

Moves the cursor up one line on the screen.

`CTRL` `[BACKSPACE]` or `[DEL]`

Deletes the character positioned at the cursor. All characters to the right then move one position left to fill in the space left by the deletion. If a logical line extends beyond one physical line, characters on subsequent lines move left one position to fill in the deleted space, and the character in the first column of each subsequent line moves up to the end of the preceding line.

`[DEL]` (delete) is the opposite of `[INS]` (insert). Deleting text reduces logical line length.

`CTRL` `[END]` or `CTRL` `[E]`

Erases from the cursor position to the end of the logical line. All physical screen lines are erased until GW-BASIC encounters the terminating `[ENTER]`.

CTRL N or **END**

Moves the cursor to the end of the logical line. Characters typed from this position are added to the line.

CTRL ENTER or **CTRL J**

Moves the cursor to the beginning of the next screen line. This lets you create logical program lines that are longer than the physical screen width. Logical lines can be up to 255 characters long. You can also use this function as a line feed.

CTRL M or **ENTER**

Enters a line into the GW-BASIC program. Also moves the cursor to the next logical line.

CTRL I or **ESC**

Erases the entire logical line on which the cursor is located.

CTRL G

Emits a beep from your computer's speaker.

CTRL K or **HOME**

Moves the cursor to the upper left corner of the screen. The screen contents are unchanged.


CTRL HOME or **CTRL L**


Clears the screen and positions the cursor in the upper left corner of the screen.

CTRL R or **INS**


Turns the Insert Mode on and off.





Insert Mode is indicated by the cursor blotting the lower half of the character position. In Graphics Mode, the normal cursor covers the whole character position. When Insert Mode is active, only the lower half of the character position is blanked by the cursor.

When Insert Mode is off, typed characters replace existing characters on the line. The space bar erases the character at the cursor position and moves the cursor one character to the right. The  key moves the cursor one character to the right without deleting any characters.

Pressing  when Insert Mode is off moves the cursor over characters to the next tab stop. Tab stops occur every eight character positions.



When Insert Mode is on, characters following the cursor move to the right as typed characters are inserted before them at the current cursor position. After each keystroke, the cursor moves one position to the right. Line wrapping is observed. That is, as characters move off the right side of the screen, they are inserted from the left on subsequent lines. Insertions increase logical line length.

When Insert Mode is on, pressing  inserts blank spaces from the current cursor position to the next tab stop. Line wrapping is observed as above.

  or  

Places the computer in a paused state. To resume operation, press any other key.

Echoes the characters you type to the printer (Lpt1:). You can press the   keys a second time to turn off the echo.

Sends the current screen contents to the printer, effectively creating a snapshot of the screen.

  or 

Moves the cursor to the next tab stop. Tab stops occur every eight columns.

Function Keys

Certain keys or combinations of keys let you perform frequently used commands or functions with a minimum number of keystrokes. These keys are called *function keys*.

You can temporarily redefine the special function keys that appear on your keyboard to meet programming requirements and specific functions of your program.

Function keys enable rapid entry of as many as 15 characters into a program by using only one keystroke. These keys are labeled F1 through F10. GW-BASIC has already assigned special functions to each of these keys. The key assignments are some of the most frequently used commands. After you load GW-BASIC, these special key functions appear on the bottom line of your screen.

Initially, the function keys are assigned the following special functions:

Table 4.1

GW-BASIC Function Key Assignments

Key	Function	Key	Function
F1	LIST	F6	,"LPT1:"†
F2	RUN†	F7	TRON†
F3	LOAD"	F8	TROFF†
F4	SAVE"	F9	KEY
F5	CONT†	F10	SCREEN 0,0,0†

Note: The † symbol following a function indicates that you needn't press ENTER after pressing the function key. GW-BASIC immediately executes the selected command.

If you choose, you can change the assignments of one or all function keys. For more information, see the explanations of the KEY and ON KEY statements in the *GW-BASIC User's Reference*.

Creating and Using Files

There are two types of files in the MS-DOS system:

- *Program files*, which contain the program or instructions for the computer
- *Data files*, which contain information used or created by program files

Program File Commands

Following are the commands and statements most frequently used with program files. The *GW-BASIC User's Reference* contains more information on each of them.

SAVE *filename* [,a][,p]

Writes the program currently residing in memory to diskette.

LOAD *filename* [,r]

Loads the program from a diskette into memory. **LOAD** deletes the current contents of memory and closes all files before loading the program.

RUN *filename* [,r]

Loads the program from a diskette into memory and runs it immediately. **RUN** deletes the current contents of memory and closes all files before loading the program.

MERGE *filename*

Loads the program from a diskette into memory but does not delete the current program already in memory.

KILL *filename*

Deletes the file from a diskette. (You can also use this command with data files.)

NAME *old filename* **AS** *new filename*

Changes the name of a diskette file. Only the **name** of the file is **changed**. The file is not modified, and it remains in the same space and position on the disk. (You can also use this command with data files.)

Data Files

GW-BASIC programs can work with two types of data files:

- Sequential files
- Random access files

Sequential files are easier to create than random access files but are limited in flexibility and speed when accessing data. GW-BASIC writes data to a sequential file as a series of ASCII characters. This data is stored, one item after another (sequentially), in the order sent. It is read back in the same way.

Creating and accessing random access files requires more program steps than creating and accessing sequential files, but random files require less room on the disk because GW-BASIC stores them in a compressed format in the form of a string.

The following sections discuss how to create and use these two types of data files.

Creating a Sequential File

The following statements and functions are used with sequential files:

CLOSE	LOF
EOF	OPEN
INPUT #	PRINT #
LINE INPUT #	PRINT # USING
LOC	UNLOCK
LOCK	WRITE #

The following program steps are required to create a sequential file and access the data in the file:

1. Open the file in output (O) mode. The current program will use this file first for output:

```
OPEN "O",#1,"filename"
```

2. Write data to the file using the PRINT# or WRITE# statement:

```
PRINT#1,A$
```

```
PRINT#1,B$
```

```
PRINT#1,C$
```

3. To access the data in the file, you must close the file and reopen it in input (I) mode:

```
CLOSE #1
```

```
OPEN "I",#1,"filename"
```

4. Use the INPUT# or LINE INPUT# statement to read data from the sequential file into the program:

```
INPUT#1,X$,Y$,Z$
```

The example that follows is a short program that creates a sequential file, named data, from information input at the terminal.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
30 IF N$="DONE" THEN END
40 INPUT "DEPARTMENT";D$
50 INPUT "DATE HIRED";H$
60 PRINT#1,N$,";",D$,";",H$
70 PRINT:GOTO 20

RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? DONE
OK
```

Accessing a Sequential File

The program in the following example accesses the file, data, created in the previous example, 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
50 CLOSE #1
RUN
EBENEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

The program in the above example reads, sequentially, every item in the file. When all the data has been read, Line 20 causes an Input past end error. To avoid this error, insert Line 15, which uses the EOF function to test for end of file:

```
15 IF EOF(1) THEN END
```

and change Line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the diskette with the PRINT# USING statement. For example, the following statement could be used to write numeric data to diskette without explicit delimiters:

```
PRINT#1,USING"####.##",";A,B,C,D
```

The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of 128-byte records that have been written to or read from the file since it was opened.

Adding Data to a Sequential File

When a sequential file is opened in O mode, the current contents are destroyed. To add data to an existing file without destroying its contents, open the file in append (A) mode.

The program in the next example can be used to create or add to a file called names. This program illustrates the use of LINE INPUT. LINE INPUT reads in characters until it sees a carriage return indicator or until it reads 255 characters. It does not stop at quotation marks or commas.

```
10 ON ERROR GOTO 2000
20 OPEN "A",#1,"NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CR EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#1,N$
170 PRINT#1,A$
180 PRINT#1,B$
190 PRINT:GOTO 120
200 CLOSE #1
2000 ON ERROR GOTO 0
```

Lines 10 and 2000 contain the ON ERROR GOTO statement. This statement enables error trapping and specifies the first Line (2000) of the error handling subroutine. Line 10 enables the error handling routine. Line 2000 disables the error handling routine and is the point to which GW-BASIC is to branch to print the error messages.

Random Access Files

Information in random access files is stored and accessed in distinct, numbered units called *records*. Since the information is called by number, the data can be called from any disk location—the program needn't read the entire disk, as when seeking sequential files, to locate data. GW-BASIC supports large random files. The maximum logical record number is $2^{32} - 1$.

You can use the following statements and functions with random files:

CLOSE	FIELD	MKI\$
CVD	LOC	MKS\$
CVI	LOCK	OPEN
CVS	LOF	PUT
EOF	LSET/RSET	UNLOCK
ET	MKD\$	

Creating a Random Access File

The following program steps are required in creating a random data file:

1. Open the file for random access (R) mode. The following example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R",#1,"filename",32
```

2. Use the FIELD statement to allocate space in the random buffer for any variables that will be written to the random file:

```
FIELD#1,20 AS N$,4 AS A$,8 AS P$
```

In the above example, the first 20 positions (bytes) in the random file buffer are allocated to the string variable N\$. The next four positions are allocated to A\$; the next eight, to P\$.

3. Use LSET or RSET to move the data into the random buffer fields in left- or right-justified format (L = left SET; R = right SET). Numeric values must be made into strings when placed in the buffer. MKI\$ converts an integer value into a string, MKS\$ converts a single-precision value, and MKD\$ converts a double-precision value.

```
LSET N$ = X$  
LSET A$ = MKS$(AMT)  
LSET P$ = TEL$
```

4. Write the data from the buffer to the diskette, using the PUT statement:

```
PUT #1, CODE%
```

The program in the following example takes information keyed as input at the terminal and writes it to a random access data file. Each time the PUT statement is executed, a record is written to the file. In the example, the two-digit CODE% input in Line 30 becomes the record number.

Note: Do not use a fielded string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of into the random file buffer.

```
10 OPEN "R", #1, "INFOFILE", 32  
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$  
30 INPUT "2-DIGIT CODE"; CODE%  
40 INPUT "NAME"; X$  
50 INPUT "AMOUNT"; AMT  
60 INPUT "PHONE"; TEL$: PRINT  
70 LSET N$ = X$  
80 LSET A$ = MKS$(AMT)  
90 LSET P$ = TEL$  
100 PUT #1, CODE%  
110 GOTO 30
```

Accessing a Random Access File

The following program steps are required to access a random file:

1. Open the file in R mode:

```
OPEN "R",#1,"filename",32
```

2. Use the FIELD statement to allocate space in the random buffer for any variables that will be read from the file:

```
FIELD, #1, 20 AS N$, 4 AS A$, 8 AS P$
```

In this example, the first 20 positions (bytes) in the random file buffer are allocated to the string variable N\$. The next four positions are allocated to A\$; the next eight, to P\$.

Note: 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.

```
GET #1,CODE%
```

The program can now access the data in the buffer.

4. Convert numeric values back to numbers using the convert functions: CVI for integers, CVS for single-precision values, and CVD for double-precision values.

```
PRINT N$  
PRINT CVS(A$)
```

```
.  
. .  
. .
```

The program in this next example accesses the random file Infofile which was created in the previous example. By inputting the three-digit code, the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"INFOFILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

With random files, the LOC function returns the current record number. The current record number is the last record number used in a GET or PUT statement. For example, the following line ends program execution if the current record number in file#1 is higher than 99:

```
IF LOC(1)>99 THEN END
```

The following example 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 that the inventory will contain no more than 100 different part numbers.

Lines 900-960 initialize the data file by writing CHR\$(255) as the first character of each record. This is used later (Line 270 and Line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type the desired function number, Line 230 branches to the appropriate subroutine.

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS
    P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER
    LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT "BAD
    FUNCTION NUMBER":GOTO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
    IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
```

```
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD";A%
520 Q% = CVI(Q$) + A%
530 LSET Q$ = MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q% = CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":
    GOTO 600
630 Q% = Q%-S%
640 IF Q% = < CVI(R$) THEN PRINT "QUANTITY NOW";
    Q%; "REORDER LEVEL";CVI(R$)
650 LSET Q$ = MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL 4
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<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
```


Constants, Variables, Expressions and Operators

After you learn the fundamentals of programming in GW-BASIC, you'll probably want to write more complex programs. The information in this chapter will help you learn more about using constants, variables, expressions, and operators in GW-BASIC, as well as how you can use them to develop more sophisticated programs.

Constants

Constants are static values the GW-BASIC Interpreter uses during execution of your program. There are two types of constants: *string* and *numeric*.

A *string constant* is a sequence of 0 to 255 alphanumeric characters enclosed in double quotation marks. The following are sample string constants:

"HELLO"

"\$25000.00"

"Number of Employees"

Numeric constants can be positive or negative. When entering a numeric constant in GW-BASIC, do not type the commas. For instance, if the number 10,000 were to be entered as a constant, it would be typed as 10000. There are five types of numeric constants: *integer*, *fixed-point*, *floating-point*, *hexadecimal*, and *octal*.

Constant	Description
Integer	Whole numbers between -32768 and +32767 that do not contain decimal points.
Fixed-Point	Positive or negative real numbers that contain decimal points.
Floating-Point	<p>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).</p> <p>The allowable range for floating-point constants is 3.0×10^{-39} to 1.7×10^{38}:</p> <p>235.988E-7 = .0000235988 2359E6 = 2359000000</p>
Hexadecimal	<p>Hexadecimal numbers with the prefix &H:</p> <p>&H76 &H32F</p>
Octal	<p>Octal numbers with the prefix &O or &:</p> <p>&O347 &1234</p>

Single-and Double-Precision Form for Numeric Constants

Numeric constants can be either integers, single-precision numbers, or double-precision numbers. Integer constants are stored as whole numbers only. Single-precision numeric constants are stored with seven digits (although only six might be accurate). Double-precision numeric constants are stored with 17 digits of precision and printed with as many as 16 digits.

A single-precision constant is any numeric constant with any of the following:

- Seven or fewer digits
- Exponential form using E
- A trailing exclamation point (!)

A double-precision constant is any numeric constant with any of the following:

- Eight or more digits
- Exponential form using D
- A trailing number sign (#)

Following are examples of single- and double-precision numeric constants:

Single-Precision	Double-Precision
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3490.0#
22.5!	7654321.1234

Variables

Variables are the names that you choose to represent values used in a GW-BASIC program. The value of a variable can be assigned specifically or can be the result of calculations in your program. If a variable is assigned no value, GW-BASIC assumes the variable's value to be zero.

Variable Names and Declarations

GW-BASIC variable names can be any length. Up to 40 characters are significant. The characters allowed in a variable name are letters, numbers, and the decimal point. The first character in the variable name must be a letter. Special *type declaration characters* are also allowed.

You cannot use reserved words as variable names. However, if the reserved word is embedded within the variable name, it is accepted. Reserved words include all the words used as GW-BASIC commands, statements, functions, and operators.

Variables can represent either numeric values or strings.

Type Declaration Characters

Type declaration characters indicate whether a variable represents a string, an integer, a single-precision number, or a double-precision number. GW-BASIC recognizes the following type declaration characters:

Character	Type of Variable	Sample
\$	String variable	N\$
%	Integer variable	LIMIT%
!	Single-precision variable	MINIMUM!
#	Double-precision variable	P1#

The default type for a numeric variable name is single-precision. Double-precision, while very accurate, uses more memory space and more calculation time. Single-precision is sufficiently accurate for most applications. However, the seventh significant digit (if printed) might not always be accurate. Be careful when making conversions among integer, single-precision, and double-precision variables.

The following variable is a single-precision value by default:

ABC

Variables beginning with FN are assumed to be calls to a user-defined function.

You can include the GW-BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL in a program to declare the types of values for certain variable names.

Array Variables

An *array* is a group or table of values referred to by the same variable name. Each *element* in an array is referred to by an *array variable* that is a subscripted integer or an integer expression. The subscript is enclosed within parentheses. An array variable name has as many subscripts as there are dimensions in the array.

For example:

V(10)

Refers to a value in a one-dimensional array.

T(1,4)

Refers to a value in a two-dimensional array.

The maximum number of dimensions for an array in GW-BASIC is 255. The maximum number of elements per dimension is 32767.

Note: If you are using an array with a subscript value greater than 10, use the DIM statement. Refer to the *GW-BASIC User's Reference* for more information. If you use a subscript greater than the maximum specified, you see the error message **Subscript out of range**.

Multidimensional arrays (more than one subscript separated by commas) are useful for storing tabular data. For example, A(1,4) could be used to represent a two-row, five-column array such as the following:

Column		0	1	2	3	4
Row	0	10	20	30	40	50
Row	1	60	70	80	90	100

In this example, element $A(1,2) = 80$ and $A(0,3) = 40$.

Rows and columns begin with 0, not 1, unless otherwise declared. For more information, see the OPTION BASE statement in the *GW-BASIC User's Reference*.

Memory Space Requirements for Variable Storage

The different types of variables require different amounts of storage. Depending on the storage and memory capacity of your computer and the size of the program that you are developing, the required storage amounts can be important considerations.

Variable Type	Required Bytes of Storage
integer	2
single-precision	4
double-precision	8

Array Type	Required Bytes of Storage
integer	2 per element
single-precision	4 per element
double-precision	8 per element

Strings: String variables require three bytes overhead, plus one byte for each character in the string. The quotation marks at the beginning and end of each string are not counted.

When necessary, GW-BASIC converts a numeric constant from one type of variable to another, according to the following rules:

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

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

```
RUN
23
```

If a string variable is set equal to a numeric value or vice versa, a **Type Mismatch** error occurs.

- During an expression evaluation, all the operands in an arithmetic or relational operation are converted to the same degree of precision, that is, that of the most precise operand. The result of an arithmetic operation is also returned to this degree of precision. For example, in the following program lines, the arithmetic is performed in double-precision, and the result is returned in D# as a double-precision value:

```
10 D# = 6#/7
20 PRINT D#
```

```
RUN
.8571428571428571
```

In the following example, the arithmetic is performed in double-precision, and the result is returned to D (single-precision variable), rounded and printed as a single-precision value.

```
10 D = 6#/7
20 PRINT D
```

- Logical operators convert their operands to integers and return an integer result. Operands must be within the range of -32768 to 32767; otherwise, an **Overflow** error occurs.

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

For example:

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

```
RUN
56
```

- If a double-precision variable is assigned a single-precision value, only the first seven digits (rounded) of the converted number are valid. This is because only seven digits of accuracy were supplied with the single-precision value. The absolute value of the difference between the printed double-precision number and the original single-precision value is less than $6.3\text{E-}8$ times the original single-precision value. For example:

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

```
RUN
2.04 2.039999961853027
```

Expressions and Operators

An *expression* can be simply a string or numeric constant, a variable, or a combination of constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by GW-BASIC are divided into four categories:

- Arithmetic
- Relational
- Logical
- Functional

Arithmetic Operators

GW-BASIC recognizes the following *arithmetic operators*. The operators appear in order of precedence.

Operator	Operation
\wedge	exponentiation
$-$	negation
$*$	multiplication
$/$	floating-point division
$+$	addition
$-$	subtraction

Operations within parentheses are performed first. Inside the parentheses, the usual order of precedence is maintained.

Following are a few sample algebraic expressions and their GW-BASIC counterparts:

Algebraic	BASIC
$\frac{X - Z}{Y}$	$(X - Y) / Z$
$\frac{XY}{Z}$	$X * Y / Z$
$\frac{X + Y}{Z}$	$(X + Y) / Z$
$(X^2)^Y$	$(X \wedge 2) \wedge Y$
X^{YZ}	$X \wedge (Y * Z)$
$X(-Y)$	$X * (-Y)$

Two consecutive operators must be separated by parentheses.

Integer Division and Modulus Arithmetic

Two additional arithmetic operators are available: *integer division* and *modulus arithmetic*.

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

The following are examples of integer division:

$$\begin{aligned}10 \backslash 4 &= 2 \\25.68 \backslash 6.99 &= 3\end{aligned}$$

In the order of occurrence within GW-BASIC, integer division is performed just after floating-point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division.

The following are examples of modulus arithmetic:

$$\begin{aligned}10.4 \text{ MOD } 4 &= 2 \\(10/4=2 \text{ with a remainder } 2) \\25.68 \text{ MOD } 6.99 &= 5 \\(26/7=3 \text{ with a remainder } 5)\end{aligned}$$

In the order of occurrence within GW-BASIC, modulus arithmetic follows integer division. As well, the INT and FIX functions, described in the *GW-BASIC User's Reference*, are also useful in modulus arithmetic.

Overflow and Division by Zero

If, during the evaluation of an expression, a division by zero is encountered, the Division by zero error message appears, your computer's representation of infinity (such as 1.701412E+38) with the sign of the numerator is supplied as the result of the division, and execution continues.

If the evaluation of an exponentiation results in zero being raised to a negative power, the Division by Zero error message appears, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the **Overflow** error message appears, machine infinity with the algebraically correct sign is supplied as the result, and execution continues. The errors that occur in overflow and division by zero are not trapped by the error trapping function.

Relational Operators

Relational operators let you compare two values. The result of the comparison is either true (-1) or false (0). This result can then be used to make a decision regarding program flow.

The following table displays the relational operators.

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

The equal sign is also used to assign a value to a variable. See the explanation of the LET statement in the *GW-BASIC User's Reference*.

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

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

This expression is true if the value of $X + Y$ is less than the value of $T-1$ divided by Z .

Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bit-wise result that is either true (not zero) or false (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following tables. The operators are listed in order of precedence.

NOT	If X is:	The Result is:	
	True	False	
	False	True	
AND	If X is:	And Y is:	Then X AND Y is:
	True	True	True
	True	False	False
	False	True	False
	False	False	False
OR	If X is:	And Y is:	Then X OR Y is:
	True	True	True
	True	False	True
	False	True	True
	False	False	False
XOR	If X is:	And Y is:	Then X XOR Y is:
	True	True	False
	True	False	True
	False	True	True
	False	False	False
EQV	If X is:	And Y is:	Then X EQV Y is:
	True	True	True
	True	False	False
	False	True	False
	False	False	True

IMP	If X is:	And Y is:	Then X IMP Y is:
	True	True	True
	True	False	False
	False	True	True
	False	False	True

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

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

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

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

The following examples demonstrate how the logical operators work:

Example	Explanation
63 AND 16 = 16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14 = 14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8 = 8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10 = 10	10 = binary 1010, so 10 OR 10 = 10 (binary 1010). -1 OR -2 = -1. -1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1 (binary 1111111111111111). The bit complement of 16 zeros is 16 ones, which is the two's complement representation of -1.
NOT X = -(X + 1)	The two's complement of any integer is the bit com- plement plus one.

Functional Operators

A *function* is used in an expression to call a predetermined operation that is to be performed on an operand. GW-BASIC has intrinsic functions that reside in the system, such as SQR (square root) or SIN (sine).

GW-BASIC also allows you to write user-defined functions. See the explanation of the DEF FN statement in the *GW-BASIC User's Reference*.

String Operators

To compare strings, use the same relational operators used with numbers:

Operator	Meaning
=	Equal to
< >	Unequal
<	Less than
>	Greater than
< =	Less than or equal to
> =	Greater than or equal to

The GW-BASIC Interpreter compares strings by taking one character at a time from each string and comparing the characters' ASCII codes. If the ASCII codes in each string are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher code. If the interpreter reaches the end of one string during string comparison, the shorter string is said to be smaller, providing that both strings are the same up to that point. Leading and trailing blanks are significant.

For example:

```
"AA" < "AB"  
"FILENAME" = "FILENAME"  
"X&" > "X#"  
"CL " > = "CL"  
"kg" > "KG"  
"SMYTH" < "SMYTHE"  
B$ < "9/12/78" where B$ = "8/12/78"
```

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

Strings can be concatenated by using the plus (+) sign. For example:

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


Error Codes and Messages

1 NEXT without FOR

A NEXT statement does not have a corresponding FOR statement. Check the FOR statement variable for a matching NEXT statement variable.

2 Syntax error

A line is encountered that contains an incorrect sequence of characters (such as unmatched parentheses, a misspelled command or statement, or incorrect punctuation). This error causes GW-BASIC to display the incorrect line in edit mode.

3 RETURN without GOSUB

A RETURN statement is encountered for which there is no previous 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. An illegal function call error can also occur as the result of:

- A negative or unreasonably large subscript
- A negative or zero argument with LOG
- A negative argument to SQR
- A negative mantissa with a non-integer power
- A call to a USR function for which the starting address has not yet been given
- 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 GW-BASIC's number format. If underflow occurs, the result is zero, and execution continues without an error.

7 Out of memory

A program is too large, has too many FOR loops, GOSUBs, variables, or expressions that are too complicated. Use the CLEAR statement to set aside more stack space or memory area.

8 Undefined line number

A line reference in a GOTO, GOSUB, IF-THEN...ELSE, or DELETE is a nonexistent line.

9 Subscript out of range

An array element is referred to 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 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 GW-BASIC to exceed the amount of free memory remaining. GW-BASIC 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. Break the expression into smaller expressions.

17 Can't continue

An attempt is made to continue a program that:

- Has halted because of an error
- Has been modified during a break in execution
- Does not exist

18 Undefined user function

A `USR` function is called before the function definition (`DEF` statement) is given.

19 No RESUME

An error-trapping routine is entered that contains no `RESUME` statement.

20 RESUME without error

A `RESUME` statement is encountered before an error-trapping routine is entered.

21 Unprintable error

No error message is available for the existing error condition. This is usually caused by an error with an undefined error code.

22 Missing operand

An expression contains an operator with no operand following it.

23 Line buffer overflow

An attempt is made to input a line that has too many characters.

24 Device Timeout

`GW-BASIC` did not receive information from an I/O device within a predetermined amount of time.

25 Device Fault

Indicates a hardware error in the printer or interface card.

26 FOR Without NEXT

A `FOR` was encountered without a matching `NEXT`.

27 Out of Paper

The printer is out of paper, or a printer fault is indicated.

28 Unprintable error

No error message is available for the existing error condition. This is usually caused by an error with an undefined error code.

29 WHILE without WEND

A WHILE statement does not have a matching WEND.

30 WEND without WHILE

A WEND was encountered without a matching WHILE.

31-49 Unprintable error

No error message is available for the existing error condition. This is usually caused by an error with an undefined error code.

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 GW-BASIC. Report to your dealer the conditions under which the message appeared.

52 Bad file number

A statement or command refers to a file with a file number that is not open or that is out of the range of file numbers specified at initialization.

53 File not found

A LOAD, KILL, NAME, FILES, or OPEN statement refers to 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 with a file mode other than I, O, A, or R.

55 File already open

A sequential output mode OPEN is issued for a file that is already open, or a KILL is given for a file that is open.

56 Unprintable error

An error message is not available for the error condition that exists. This is usually caused by an ERROR with an undefined error code.

57 Device I/O Error

Usually a disk I/O error, but generalized to include all I/O devices. This is a fatal error; that is, the operating system cannot recover from the error.

59-60 Unprintable error

No error message is available for the existing error condition. This is usually caused by an error with an undefined error code.

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 is used for the filename with LOAD, SAVE, KILL, or OPEN. Example: a filename contains too many characters.

65 Unprintable error

No error message is available for the existing error condition. This is usually caused by an error with an undefined error code.

66 Direct statement in file

A direct statement is encountered while loading an ASCII-format file. The LOAD is terminated.

67 Too many files

An attempt is made to create a new file (using SAVE or OPEN) when all directory entries are full or the file specifications are invalid.

68 Device Unavailable

An attempt is made to open a file to a nonexistent device. It might be that hardware does not exist to support the device, such as lpt2: or lpt3:, or that it is disabled by the user. This occurs if an OPEN "COM1: statement is executed but the user disables RS232 support with the /c: switch directive on the command line.

69 Communication buffer overflow

Occurs when a communications input statement is executed when the input queue is already full. Use an ON ERROR GOTO statement to retry the input when this condition occurs. Subsequent inputs attempt to clear this fault unless characters continue to be received faster than the program can process them. In this case, several options are available:

- Increase the size of the COM receive buffer by using the /c: switch.
- Implement a hand-shaking protocol with the host/satellite (such as: XON/XOFF, as demonstrated in the TTY programming example) to turn transmit off long enough to catch up.
- Use a lower baud rate for transmit and receive.

70 Permission Denied

This is one of three hard disk errors returned from the diskette controller.

- An attempt has been made to write onto a diskette that is write protected.
- Another process has attempted to access a file already in use.
- The UNLOCK range specified does not match the preceding LOCK statement.

71 Disk not Ready

Occurs when the diskette drive door is open or a diskette is not in the drive. Use an ON ERROR GOTO statement to recover.

72 Disk media error

Occurs when the diskette controller detects a hardware or media fault. This usually indicates damaged media. Copy any existing files to a new diskette, and reformat the damaged diskette. FORMAT maps the bad tracks in the file allocation table. The remainder of the diskette is now usable.

73 Advanced Feature

An attempt was made to use a reserved word that is not available in this version of GW-BASIC.

74 Rename across disks

An attempt is made to rename a file to a new name declared to be on a disk other than the disk specified for the old name. The naming operation is not performed.

75 Path/File Access Error

During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS is 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, MS-DOS is unable to find the path specified. The operation is not completed.

Appendix B

ASCII Character Codes

ASCII Code	Character	Control Character
000	(null)	NUL
001	☺	SOH
002	☹	STX
003	♥	ETX
004	♦	EOT
005	♣	ENQ
006	♠	ACK
007	(beep)	BEL
008	■	BS
009	(tab)	HT
010	(line feed)	LF
011	(home)	VT
012	(form feed)	FF
013	(carriage return)	CR
014	🎵	SO
015	⚙	SI
016	▶	DLE
017	◀	DC1
018	↕	DC2
019	!!	DC3
020	¶	DC4
021	§	NAK
022	—	SYN
023	⤵	ETB
024	↑	CAN
025	↓	EM
026	→	SUB
027	←	ESC
028	(cursor right)	FS
029	(cursor left)	GS
030	(cursor up)	RS
031	(cursor down)	US

ASCII Code	Character	ASCII Code	Character
032	(space)	068	D
033	!	069	E
034	''	070	F
035	#	071	G
036	\$	072	H
037	%	073	I
038	&	074	J
039	'	075	K
040	(076	L
041)	077	M
042	*	078	N
043	+	079	O
044	,	080	P
045	-	081	Q
046	.	082	R
047	/	083	S
048	0	084	T
049	1	085	U
050	2	086	V
051	3	087	W
052	4	088	X
053	5	089	Y
054	6	090	Z
055	7	091	[
056	8	092	\
057	9	093]
058	:	094	^
059	;	095	_
060	<	096	`
061	=	097	a
062	>	098	b
063	?	099	c
064	@	100	d
065	A	101	e
066	B	102	f
067	C	103	g

ASCII Code	Character	ASCII Code	Character
104	h	140	î
105	i	141	ï
106	j	142	Ä
107	k	143	Å
108	l	144	É
109	m	145	æ
110	n	146	Æ
111	o	147	ô
112	p	148	Ö
113	q	149	ò
114	r	150	û
115	s	151	ù
116	t	152	ÿ
117	u	153	Ö
118	v	154	Ü
119	w	155	¢
120	x	156	£
121	y	157	¥
122	z	158	Pt
123	{	159	f
124		160	á
125	}	161	í
126	~	162	ó
127	☐	163	ú
128	Ç	164	ñ
129	ü	165	Ñ
130	é	166	à
131	â	167	o
132	ä	168	z
133	à	169	┌
134	å	170	└
135	ç	171	½
136	ê	172	¼
137	ë	173	ì
138	è	174	«
139	ï	175	»

ASCII Code	Character	ASCII Code	Character
176	⚡	212	ℓ
177	⚡	213	ℓ
178	⚡	214	ℓ
179	⚡	215	ℓ
180	⚡	216	ℓ
181	⚡	217	ℓ
182	⚡	218	ℓ
183	⚡	219	■
184	⚡	220	■
185	⚡	221	■
186	⚡	222	■
187	⚡	223	■
188	⚡	224	α
189	⚡	225	β
190	⚡	226	Γ
191	⚡	227	π
192	⚡	228	Σ
193	⚡	229	σ
194	⚡	230	μ
195	⚡	231	τ
196	⚡	232	Φ
197	⚡	233	Θ
198	⚡	234	Ω
199	⚡	235	δ
200	⚡	236	∞
201	⚡	237	Ø
202	⚡	238	(
203	⚡	239	∩
204	⚡	240	≡
205	⚡	241	±
206	⚡	242	≥
207	⚡	243	≤
208	⚡	244	∫
209	⚡	245	∫
210	⚡	246	-
211	⚡	247	≈

ASCII Code	Character	ASCII Code	Character
248	°	252	η
249	•	253	²
250	•	254	■
251	√	255	(blank 'FF')

Extended Codes

For certain keys and key combinations, INKEY\$ returns a 2-character code. The first character is a null character (ASCII Code 00). The second is usually the scan code of the key(s) pressed. The key(s) and associated ASCII codes (in decimal) are listed below.

Second Character	Key(s) Pressed	Second Character	Key(s) Pressed
15	SHIFT TAB	64	F6
16	ALT Q	65	F7
17	ALT W	66	F8
18	ALT E	67	F9
19	ALT R	68	F10
20	ALT T	71	HOME
21	ALT Y	72	↑
22	ALT U	73	PG UP
23	ALT I	75	←
24	ALT O	77	→
25	ALT P	79	END
30	ALT A	81	PG DN
31	ALT S	82	INSERT
32	ALT D	83	DELETE
33	ALT F	84	SHIFT F1
34	ALT G	85	SHIFT F2
35	ALT H	86	SHIFT F3
36	ALT J	87	SHIFT F4
37	ALT K	88	SHIFT F5
38	ALT L	89	SHIFT F6
44	ALT Z	90	SHIFT F7
45	ALT X	91	SHIFT F8
46	ALT C	92	SHIFT F9
47	ALT V	93	SHIFT F10
48	ALT B	94	CTRL F1
49	ALT N	95	CTRL F2
50	ALT M	96	CTRL F3
59	F1	97	CTRL F4
60	F2	98	CTRL F5
61	F3	99	CTRL F6
62	F4	100	CTRL F7
63	F5	101	CTRL F8

Second Character	Key(s) Pressed	Second Character	Key(s) Pressed
102	CTRL F9	118	CTRL PG DN
103	CTRL F10	119	CTRL HOME
104	ALT F1	120	ALT 1
105	ALT F2	121	ALT 2
106	ALT F3	122	ALT 3
107	ALT F4	123	ALT 4
108	ALT F5	124	ALT 5
109	ALT F6	125	ALT 6
110	ALT F7	126	ALT 7
111	ALT F8	127	ALT 8
112	ALT F9	128	ALT 9
113	ALT F10	129	ALT 0
115	CTRL ←	130	ALT -
116	CTRL →	131	ALT =
117	CTRL END	132	CTRL PG UP

Assembly Language (Machine Code) Subroutines

This appendix is written primarily for users experienced in assembly language programming.

GW-BASIC lets you interface with assembly language subroutines by using the USR function and the CALL statement. USR allows BASIC to call assembly language subroutines in the same way it calls GW-BASIC intrinsic functions. However, we recommend the CALL statement for interfacing machine language programs with GW-BASIC. The CALL statement is compatible with more languages than is the USR function call, produces more readable source code, and can pass multiple arguments.

Memory Allocation

Memory space must be set aside for an assembly language (or a machine code) subroutine before it can be loaded. There are three recommended ways to set aside space for assembly language routines:

- Specify an array and use VARPTR to locate the start of the array before every access.
- Use the /m switch in the command line. Get GW-BASIC's Data segment (DS), and add the size of DS to refer to the reserved space above the data segment.
- Execute a .COM file that stays resident, and store a pointer to it in an unused interrupt vector location.

There are three recommended ways to load assembly language routines:

- **BLOAD** the file. Use **DEBUG** to load in a .EXE file that is in high memory, run GW-BASIC, and **BSAVE** the .EXE file.
- Execute a .COM file that contains the routines. Save the pointer to these routines in unused interrupt-vector locations so that your application in GW-BASIC can get the pointer and use the routine(s).
- Place the routine into the specified area.

If more stack space is needed when an assembly language subroutine is called, GW-BASIC stack space can be saved, and a new stack set up for use by the assembly language subroutine. The GW-BASIC stack space must be restored, however, before returning from the subroutine.

CALL Statement

CALL *variablename*[(*arguments*)]

variablename contains the offset in the current segment of the subroutine being called.

arguments are the variables or constants, separated by commas, that are to be passed to the routine.

For each parameter in arguments, the two-byte offset of the parameter's location within the data segment (DS) is pushed onto the stack.

The GW-BASIC return address code segment (CS) and offset (IP) are pushed onto the stack.

A long call to the segment address given in the last DEF SEG statement and the offset given in *variablename* transfers control to your routine.

The stack segment (SS), data segment (DS), extra segment (ES), and stack pointer (SP) must be preserved.

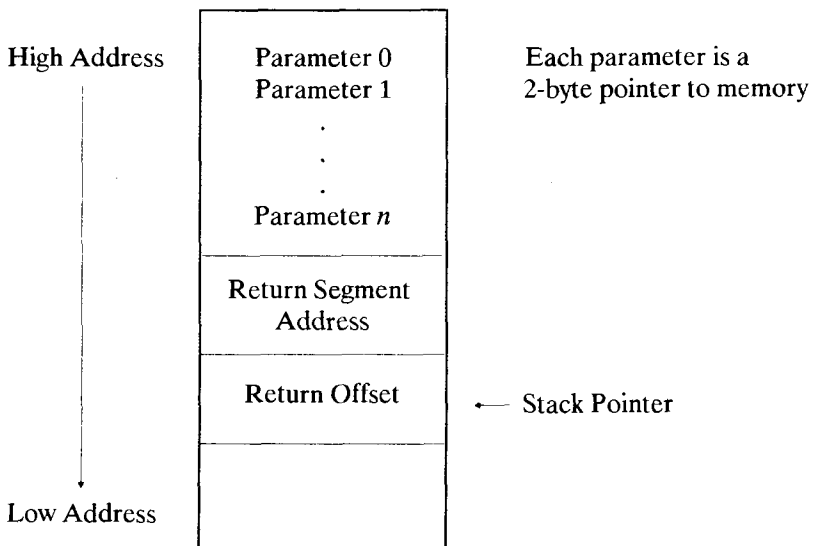


Figure D.1 Stack Layout When the CALL Statement is Activated

Figure D.1 shows the state of the stack at the time of the CALL statement:

Your routine now has control. Parameters can be referred to by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP. Upon entry, the segment registers DS, ES, and SS all point to the address of the segment that contains the GW-BASIC interpreter code. The code segment register CS contains the latest value supplied by DEF SEG. If no DEF SEG has been specified, it then points to the same address as DS, ES, and SS (the default DEF SEG).

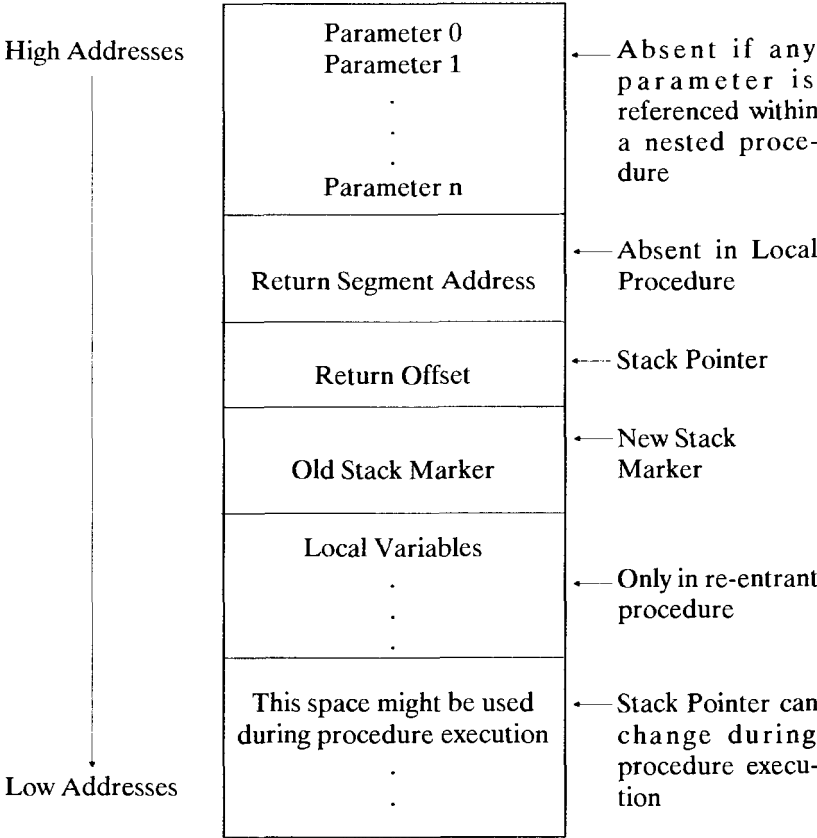


Figure D.2 Stack Layout During Execution of a CALL Statement

Figure D.2 shows the condition of the stack during execution of the called subroutine:

You must observe the following rules when coding a subroutine:

- The called routine can destroy the contents of the AX, BX, CX, DX, SI, DI, and BP registers. They do not require restoration upon return to GW-BASIC. However, all segment registers and the stack pointer must be restored. Good programming practice dictates that interrupts enabled or disabled be restored to the state observed upon entry.
- The called program must know the number and length of the parameters passed. References to parameters are positive offsets added to BP, assuming the called routine moved the current stack pointer into BP; that is, MOV BP,SP. When three parameters are passed, the location of P0 is at BP + 10, P1 is at BP + 8, and P2 is at BP + 6.
- The called routine must perform a RETURN *n* (*n* is two times the number of parameters in the argument list) to adjust the stack to the start of the calling sequence. Also, programs must be defined by a PROC FAR statement.
- Values are returned to GW-BASIC by including in the argument list the variable name that receives the result.
- If the argument is a string, the parameter offset points to three bytes 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 eight bits of the string starting address in string space.

Note: The called routine must not change the contents of any of the three bytes of the string descriptor.

- Your routines can alter strings, but they are not allowed to change the strings' lengths. GW-BASIC cannot correctly manipulate strings if their lengths are modified by external routines.
- If the argument is a string literal in the program, the string descriptor points 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, the following line forces the string literal to be copied into string space allocated outside of program memory space:

```
20 A$ = "BASIC" + ""
```

The string can then be modified without affecting the program.

Examples:

```
100 DEF SEG = &H2000
110 ACC = &H7FA
120 CALL ACC(A,B$,C)
```

·
·
·

In the above example, Line 100 sets the segment to 2000 hex. The value of variable ACC is added into the address as the low word after the DEF SEG value is left-shifted four bits. (This is a function of the microprocessor, not of GW-BASIC.) Here, ACC is set to &H7FA, so that the call to ACC executes the subroutine at location 2000:7FA hex.

Upon entry, only 16 bytes (eight words) remain available within the allocated stack space. If the called program requires additional stack space, then your program must reset the stack pointer to a new allocated space. Be sure to restore the stack pointer adjusted to the start of the calling sequence on return to GW-BASIC.

The following example demonstrates access of the parameters passed and storage of a return result in the variable C.

Note: The called program must know the variable type for numeric parameters passed. In these examples, the following instruction copies only two bytes:

MOVSW

This is adequate if variables A and C are integer. It would be necessary to copy four bytes if they were single precision or eight bytes if they were double precision.

MOV BP,SP	Gets the current stack position in BP
MOV BX,8[BP]	Gets the address of B\$ description
MOV CL,[BX]	Gets the length of B\$ in CL
MOV DX,1[BX]	Gets the address of B\$ string descriptor in DX
MOV SI,10[BP]	Gets the address of A in SI
MOV DI,6[BP]	Gets the pointer to C in DI
MOVSW	Stores variable A in 'C'
RET 6	Restores stack; returns

USR Function Calls

Although the CALL statement is the recommended way of calling assembly language subroutines, the USR function call is still available for compatibility with previously written programs.

Syntax:

USR[*n*](*argument*)

n is a number, 0-9, that specifies the USR routine being called. (See the DEF USR statement.) If *n* is omitted, USR0 is assumed.

argument is any numeric or string expression.

In GW-BASIC, a DEF SEG statement should be executed prior to a USR function call to ensure 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.

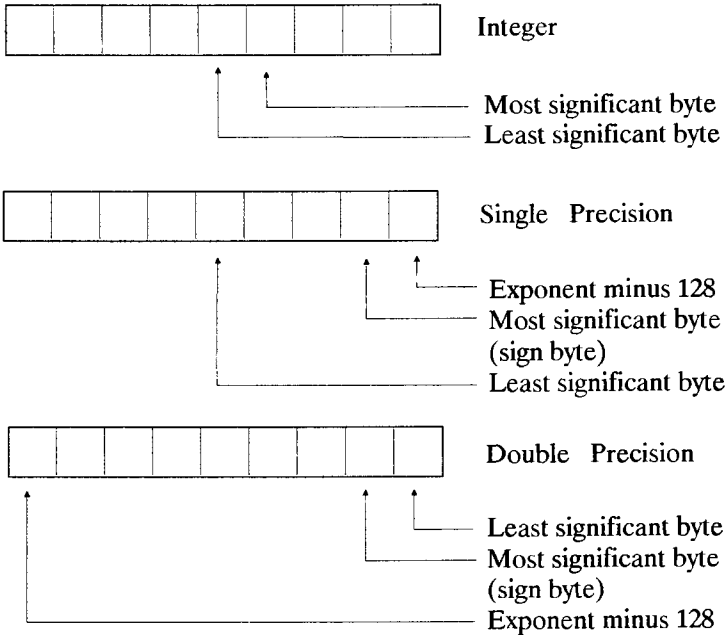
For each USR function call, a corresponding DEF USR statement must have been executed to define the USR function call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register AL contains the *number type flag* (NTF), which specifies the type of argument given. The NTF value can be one of the following:

NTF Value	Specifies
-----------	-----------

2	a two-byte integer (two's complement format)
3	a string
4	a single-precision floating point number
8	a double-precision floating point number

If the argument of a USR function call is a number ($AL < > 3$), the value of the argument is placed in the *floating-point accumulator* (FAC). The FAC is eight bytes long and is in the GW-BASIC data segment. Register BX will point at the fifth byte of the FAC. Figure D.3 shows the representation of all the GW-BASIC number types in the FAC:



Number Types in Floating-Point Accumulator

If the argument is a single-precision, floating-point number:

- $BX + 3$ is the exponent, minus 128. The binary point is to the left of the most significant bit of the mantissa.
- $BX + 2$ contains the highest seven bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number. (0 = positive, 1 = negative.)
- $BX + 1$ contains the middle eight bits of the mantissa.
- $BX + 0$ contains the lowest eight bits of the mantissa.

If the argument is an integer:

- BX + 1 contains the upper eight bits of the argument.
- BX + 0 contains the lower eight bits of the argument.

If the argument is a double-precision, floating-point number:

- BX + 0 through BX + 3 are the same as for single-precision, floating-point.
- BX-1 to BX-4 contain four more bytes of mantissa. BX-4 contains the lowest eight bits of the mantissa.

If the argument is a string (indicated by the value 3 stored in the AL register):

The (DX) register pair points to three bytes 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 eight bits of the string starting address in the GW-BASIC data segment.

If the argument is a string literal in the program, the string descriptor points to program text. Be careful not to alter or destroy programs this way. (See the preceding CALL statement.)

Usually, the value returned by a USR function call is the same type (integer, string, single-precision, or double-precision) as the argument that was passed to it. The registers that must be preserved are the same as in the CALL statement.

A far return is required to exit the USR subroutine. The returned value must be stored in the FAC.

Programs That Call Assembly Language Programs

This section contains two sample GW-BASIC programs that:

- load an assembly language routine to add two numbers together
- return the sum into memory
- remain resident in memory

The code segment and offset to the first routine is stored in the interrupt vector at 0:100H.

The following example calls an assembly language subroutine:

```
10 DEF SEG=0
100 CS=PEEK(&H102)+PEEK(&H103)*256
200 OFFSET=PEEK(&H100)+PEEK(&H101)*256
250 DEF SEG
300 C1%=2:C2%=3:C3%=0
400 TWOSUM=OFFSET
500 DEF SEG=CS
600 CALL TWOSUM(C1%,C2%,C3%)
700 PRINT C3%
800 END
```

The assembly language subroutine called in the above program must be assembled, linked, and converted to a .COM file. The program, when executed prior to the running of the GW-BASIC program, will remain in memory until the system power is turned off or the system is rebooted.

```
0100          org 100H
0100          double segment
                assume cs:double
0100 EB 17 90  start: jmp start1
0103          usrpgr proc far
0103 55          push bp
0104 8B EC          mov bp,sp
0106 8B 76 08       mov si,[bp]+8      ;get address of
                                ;parameter b
0109 8B 04          mov ax,[si]        ;get value of b
010B 8B 76 0A       mov si,[bp]+10     ;get address of
                                ;parameter a
010E 03 04          add ax,[si]        ;add value of a to
                                ;value of b
0110 8B 7E 06       mov di,[bp]+6     ;get address of
                                ;parameter c
0113 89 05          mov [di],ax        ;store sum in
                                ;parameter c
0115 5D          pop bp
0116 ca 0006       ret 6
0119          usrpgr endp
```



```
                                ;
                                ;Prog. put procedure
                                ;in memory and remain
                                ;resident. The offset and
                                ;segment are stored in
                                ;location 100-103H.

0119                                start1:
0119 B8 0000                        mov ax,0
011C 8E D8                          mov ds,ax                ;data segment to 0000H
011E BB 0100                        mov bx,0100H            ;pointer to int vector 40H
0121 83 3F 00                      cmp word ptr [bx],0
0125 75 16                          jne quit                ;program
                                                ;already run, exit

0127 83 7F 02 00                  cmp word ptr [bx+2],0
012A 75 11                          jne quit                ;program
                                                ;already run, exit

012C B8 0103 R                    mov ax,offset usrprg
012F 89 07                          mov [bx],ax            ;program offset
0131 8C c8                          mov ax,cs
0133 89 47 02                      mov [bx+2],ax        ;data segment
0136 0E                            push cs
0137 1F                            pop ds
0138 BA 0119 R                    mov dx,offset start1
013B CD 27                        int 27h
013D                                quit:
013D CD 20                        int 20h
013F                                double ends
                                end start
```

The following example places the assembly language subroutine in the specified area:

```
10 I=0:JC=0
100 DIM A%(23)
150 MEM%=VARPTR(A%(1))
200 FOR I=1 TO 23
300 READ JC
400 POKE MEM%,JC
450 MEM%=MEM%+1
500 NEXT
600 C1%=2:C2%=3:C3%=0
700 TWOSUM=VARPTR(A%(1))
800 CALL TWOSUM(C1%,C2%,C3%)
900 PRINT C3%
950 END
1000 DATA &H55,&H8b,&Hec &H8b,&H76,&H08,&H8b,
        &H04,&H8b,&H76
1100 DATA &H0a,&H03,&H04,&H8b,&H7e,&H06,&H89,&H05,
        &H5d
1200 DATA &Hca,&H06,&H00
```

Converting BASIC Programs to GW-BASIC

Programs written in a BASIC language other than GW-BASIC might require some minor adjustments before they can be run. This appendix describes these adjustments.

String Dimensions

Delete all statements used to declare the length of strings. A statement such as the following:

```
DIM A$(I,J)
```

which dimensions a string array for J elements of length I, should be converted to the following statement:

```
DIM A$(J)
```

Some BASIC languages use a comma or ampersand (&) for string concatenation. Each of these must be changed to a plus sign (+), which is the operator for GW-BASIC string concatenation.

In GW-BASIC, the MID\$, RIGHT\$, and LEFT\$ functions are used to take substrings of strings. Forms such as A\$(I) to access the Ith character in A\$, or A\$(I,J) to take a substring of A\$ from position I to position J, must be changed as follows:

Other BASIC:	GW-BASIC:
X\$ = A\$(I)	X\$ = MID\$(A\$,I,1)
X\$ = A\$(I,J)	X\$ = MID\$(A\$,I,J-I + 1)

If the substring reference is on the left side of an assignment and X\$ is used to replace characters in A\$, convert as follows:

Other BASIC:	GW-BASIC:
A\$(I) = X\$	MID\$(A\$,I,1) = X\$
A\$(I,J) = X\$	MID\$(A\$,I,J-I + 1) = X\$

Multiple Assignments

Some BASIC languages allow statements of the following form to set B and C equal to zero:

```
10 LET B = C = 0
```

GW-BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Convert this statement to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASIC languages use a backslash (\) to separate multiple statements on a line. With GW-BASIC, be sure all elements on a line are separated by a colon (:).

MAT Functions

To execute properly, programs using the MAT functions available in some BASIC languages must be rewritten using FOR-NEXT loops.

FOR-NEXT Loops

Some BASIC languages will always execute a FOR-NEXT loop once, regardless of the limits. GW-BASIC checks the limits first and does not execute the loop if it is past the limits.

Communications

This appendix describes the GW-BASIC statements necessary to support RS-232 asynchronous communications with other computers and peripheral devices.

Opening Communications Files

The OPEN COM statement allocates a buffer for input and output in the same manner as the OPEN statement opens disk files.

Communications I/O

Since the communications port is opened as a file, all I/O statements valid for disk files are valid for COM.

COM sequential input statements are the same as those for disk files:

INPUT#
LINE INPUT#
INPUT\$

COM sequential output statements are the same as those for diskette:

PRINT#
PRINT# USING

See the *GW-BASIC User's Reference* for more information on these statements.

The COM I/O Functions

The most difficult aspect of asynchronous communications is processing characters as quickly as they are received. At rates above 2400 baud (bps), it is necessary to suspend character transmission from the host long enough for the receiver to catch up. This can be done by sending XOFF (CTRL-S) to the host to temporarily suspend transmission, and XON (CTRL-Q) to resume, if the application supports it.

GW-BASIC provides three functions that help determine when an over-run condition is imminent:

LOC(x) Returns the number of characters in the input queue waiting to be read. The input queue can hold more than 255 characters (determined by the /c: switch). If there are more than 255 characters in the queue, LOC(x) returns 255. Since a string is limited to 255 characters, this practical limit alleviates the need for you to test for string size before reading data into it.

LOF(x) Returns the amount of free space in the input queue, that is:

/c:(size)-number of characters in the input queue

LOF(x) can be used to detect when the input queue is reaching storage capacity.

EOF(x) True (-1) is returned if the input queue is empty. False (0) is returned if any characters are waiting to be read.

Possible Errors

A Communications buffer overflow error occurs if a read is attempted after the input queue is full (if LOC(x) returns 0).

A Device I/O error occurs if any of the following line conditions are detected on receive: overrun error (OE), framing error (FE), or break interrupt (BI). The error is reset by subsequent inputs, but the character causing the error is lost.

A Device fault error occurs if data set ready (DSR) is lost during I/O.

A Parity error occurs if the PE (parity enable) option was used in the OPEN COM statement and incorrect parity was received.

The INPUT\$ Function

The INPUT\$ function is preferred over the INPUT and LINE INPUT statements for reading COM files because all ASCII characters might be significant in communications. INPUT is least desirable because input stops when a comma or an enter is seen. LINE INPUT terminates when an ENTER is seen.

INPUT\$ allows all characters read to be assigned to a string.

INPUT\$ returns *x* characters from the *y* file. The following statements, then, are most efficient for reading a COM file:

```
10 WHILE NOT EOF(1)
20 A$=INPUT$(LOC(1),#1)
30 ...
40 ... Process data returned in A$ ...
50 ...
60 WEND
```

This sequence of statements translates: As long as something is in the input queue, return the number of characters in the queue, and store them in A\$. If there are more than 255 characters, only 255 are returned at a time to prevent string overflow. If this is the case, EOF(1) is false, and input continues until the input queue is empty.

GET and PUT Statements for COM Files

Purpose:

To allow fixed-length I/O for COM.

Syntax:

GET *filename*, *nbytes* PUT *filename*, *nbytes*

Comments:

filename is an integer expression returning a valid file number.

nbytes is an integer expression returning the number of bytes to be transferred into or out of the file buffer. *nbytes* cannot exceed the value set by the /s: switch when GW-BASIC was invoked.

Because of the low performance associated with telephone line communications, we recommend that you not use GET and PUT in such applications.

The following TTY sample program is an exercise in communications I/O. It is designed to enable your computer to be used as a conventional terminal. Besides full-duplex communications with a host, the TTY program allows data to be downloaded to a file as well as uploaded (transmitted) to another machine.

In addition to demonstrating the elements of asynchronous communications, this program is useful for transferring GW-BASIC programs and data to and from a computer.

Note: This program is set up to communicate with a DEC® SYSTEM-20 especially in the use of XON and XOFF. It might require modification to communicate with other types of hardware.


```
10 SCREEN 0,0:WIDTH 80
15 KEY OFF:CLS:CLOSE
20 DEFINT A-Z
25 LOCATE 25,1
30 PRINT STRING$(60," ")
40 FALSE=0:TRUE=NOT FALSE
50 MENU=5 'Value of MENU Key (^E)
60 XOFF$=CHR$(19):XON$=CHR$(17)
100 LOCATE 25,1:PRINT "Async TTY Program";
110 LOCATE 1,1:LINE INPUT "Speed?";SPEED$
120 COMFIL$="COM1:" + SPEED$ + ",E,7"
130 OPEN COMFIL$ AS #1
140 OPEN "SCRN:"FOR OUTPUT AS #3
200 PAUSE=FALSE
210 A$=INKEY$:IF A$=""THEN 230
220 IF ASC(A$)=MENU THEN 300 ELSE PRINT #1,A$;
230 IF EOF(1) THEN 210
240 IF LOC(1)>128 THEN PAUSE=TRUE:PRINT #1,XOFF$;
250 A$=INPUT$(LOC(1),#1)
260 PRINT #3,A$;:IF LOC(1)>0 THEN 240
270 IF PAUSE THEN PAUSE=FALSE:PRINT #1,XON$;
280 GOTO 210
300 LOCATE 1,1:PRINT STRING$(30,32):LOCATE 1,1
310 LINE INPUT "FILE?";DSKFIL$
400 LOCATE 1,1:PRINT STRING$(30,32):LOCATE 1,1
410 LINE INPUT"(T)ransmit or (R)eceive?";TXRX$
420 IF TXRX$="T" THEN OPEN DSKFIL$ FOR INPUT AS
    #2:GOTO 1000
430 OPEN DSKFIL$ FOR OUTPUT AS #2
440 PRINT #1,CHR$(13);
500 IF EOF(1) THEN GOSUB 600
510 IF LOC(1)>128 THEN PAUSE=TRUE:PRINT #1,XOFF$;
520 A$=INPUT$(LOC(1),#1)
530 PRINT #2,A$;:IF LOC(1)>0 THEN 510
540 IF PAUSE THEN PAUSE=FALSE:PRINT #1,XON$;
550 GOTO 500
600 FOR I=1 TO 5000
```

```
610 IF NOT EOF(1) THEN I=9999
620 NEXT I
630 IF I >= 9999 THEN RETURN
640 CLOSE #2:CLS:LOCATE 25,10:PRINT "*" Download
    complete *";
650 RETURN 200
1000 WHILE NOT EOF(2)
1010 A$=INPUT$(1,#2)
1020 PRINT #1,A$;
1030 WEND
1040 PRINT #1,CHR$(28); ^Z to make close file.
1050 CLOSE #2:CLS:LOCATE 25,10:PRINT "*** Upload
    complete ***";
1060 GOTO 200
9999 CLOSE:KEY ON
```

Notes on the TTY Sample Program

Note: *Asynchronous* implies character I/O as opposed to line or block I/O. Therefore, all prints (either to the COM file or to the screen) are terminated with a semicolon (;). This slows the return line feed normally issued at the end of the PRINT statement.

Line Number	Comments
10	Sets the SCREEN to black and white alpha mode, and sets the width to 80.
15	Turns off the soft key display, clears the screen, and assures that all files are closed.
20	Defines all numeric variables as integer, primarily for the benefit of the subroutine at 600-620. Any program looking for speed optimization should use integer counters in loops where possible.
40	Defines Boolean true and false.
50	Defines the ASCII (ASC) value of the MENU key.
60	Defines the ASCII XON and XOFF characters.
100-130	Prints program ID and asks for baud rate (speed). Opens communications to file number 1, even parity, 7 data bits.

- 200-280 This section performs full-duplex I/O between the video screen and the device connected to the RS-232 connector as follows:
1. Read a character from the keyboard into A\$. INKEY\$ returns a null string if no character is waiting.
 2. If a keyboard character is available, waiting, then:
 If the character is the MENU key, the operator is ready to download a file. Get filename.
 If the character (A\$) is not the MENU key, send it by writing to the communications file (PRINT #1...).
 3. If no character is waiting, check to see whether any characters are being received.
 4. At 230, see whether any characters are waiting in COM buffer. If not, go back and check the keyboard.
 5. At 240, if more than 128 characters are waiting, set PAUSE flag to indicate that input is being suspended. Send XOFF to host, stopping further transmission.
 6. At 250-260, read and display contents of COM buffer on screen until empty. Continue to monitor size of COM buffer (in 240). Suspend transmission if reception falls behind.
 7. Resume host transmission by sending XON only if suspended by previous XOFF.
 8. Repeat process until the MENU key is pressed.
- 300-310 Gets disk filename to be downloaded to or uploaded from.
- 400-430 Asks whether file named is to be transmitted (uploaded) or received (down-loaded).

- 440 Receive routine. Sends a RETURN to the host to begin the download. This program assumes that the last command sent to the host was to begin such a transfer and was missing only the terminating RETURN. If a DEC[®] system is the host, such a command might be:
- COPY TTY: = MANUAL.MEM (MENU Key)
- if the MENU key were pressed instead of RETURN.
- 500 When no more characters are being received, EOF(1) returns 0, and the program jumps to the timeout routine at Line 600.
- 510 If more than 128 characters are waiting, signals a pause and sends XOFF to the host.
- 520-530 Reads all characters in COM queue (LOC(x)) and writes them to diskette (PRINT #2...) until reception catches up to transmission.
- 540-550 If a pause is issued, restarts host by sending XON and clearing the pause flag. Continues the process until no characters are received for a predetermined time.
- 600-650 Time-out subroutine. The FOR loop count was determined by experimentation. If no character is received from the host for 17-20 seconds, transmission is assumed complete. If any character is received during this time (Line 610), then sets *n* well above the FOR loop range to exit loop and return to caller. If host transmission is complete, closes the disk file and resumes regular activities.
- 1000-1060 Transmit routine. Until end of disk file, reads one character into A\$ with INPUT\$ statement. Sends character to COM device in 1020. Sends a ^Z at end of file in 1040 in case receiving device needs one to close its file. Lines 1050 and 1060 close disk file, print completion message, and go back to conversation mode in Line 200.
- 9999 Presently not executed. As an exercise, add some lines to the routine 400-420 to exit the program via Line 9999. This line closes the COM file left open and restores the function key display.
-

Hexadecimal Equivalents

Table G.1 lists decimal and binary equivalents to hexadecimal values.

Table G.1
Decimal and Binary Equivalents to Hexadecimal Values

Hexadecimal Value:	Decimal Equivalent:	Binary Equivalent:
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Table G.2 lists decimal equivalents to hexadecimal values.

Table G.2
Decimal Equivalents to Hexadecimal Values

Hexadecimal Value	Decimal Equivalent:	Hexadecimal Value:	Decimal Equivalent:
0	0	80	128
1	1	.	
2	2	.	
3	3	.	
4	4	90	144
5	5	.	
6	6	.	
7	7	.	
8	8	A0	160
9	9	.	
A	10	.	
B	11	.	
C	12	B0	176
D	13	.	
E	14	.	
F	15	.	
10	16	C0	192
11	17	.	
12	18	.	
13	19	.	
14	20	D0	208
15	21	.	
16	22	.	
17	23	.	
18	24	E0	224
19	25	.	
1A	26	.	
1B	27	.	
1C	28	F0	240
1D	29	100	256
1E	30	200	512
1F	31	300	768
20	32	400	1024
.	.	500	1280

Hexadecimal Value	Decimal Equivalent:	Hexadecimal Value:	Decimal Equivalent:
.	.	600	1536
.	.	700	1792
30	48	800	2048
.	.	900	2304
.	.	A00	2560
.	.	B00	2816
40	64	C00	3072
.	.	D00	3328
.	.	E00	3584
.	.	F00	3840
50	80	1000	4096
.	.	2000	8192
.	.	3000	12288
.	.	4000	16384
60	96	5000	20480
.	.	6000	24576
.	.	7000	28672
.	.	8000	32768
70	112	9000	36864
.	.	A000	40960
.	.	B000	45056
.	.	C000	49152
.	.	D000	53248
.	.	E000	57344
.	.	F000	61440

Appendix G

Key Scan Codes

key # - SCAN CODE	NORM CASE (ASCII code)		UPPER CASE (ASCII code)		CTRL CASE (ASCII code)		ALT CASE (ASCII code)	
01	ESC	1B	ESC	1B	ESC	1B	—	—
02	1	31	!	21	—	—	ALT1	X078
03	2	32	@	40	NULL	00	ALT2	X079
04	3	33	#	23	—	—	ALT3	X07A
05	4	34	\$	24	—	—	ALT4	X07B
06	5	35	%	25	—	—	ALT5	X07C
07	6	36	-	5E	RS	1E	ALT6	X07D
08	7	37	&	26	—	—	ALT7	X07E
09	8	38	*	2A	—	—	ALT8	X07F
0A	9	39	(28	—	—	ALT9	X080
0B	0	30)	29	—	—	ALT0	X081
0C	-	2D	_	5F	US	1F	ALT-	X082
0D	=	3D	+	2B	—	—	ALT=	X083
0E	BS	08	BS	08	DEL	7F	—	—
0F	-	09	-	X00F	—	—	—	—
10	q	71	Q	51	DC1	11	ALTQ	X010
11	w	77	W	57	ETB	17	ALTW	X011
12	e	65	E	45	ENQ	05	ALTE	X012
13	r	72	R	52	DC2	12	ALTR	X013
14	t	74	T	54	DC4	14	ALTT	X014
15	y	79	Y	59	EM	19	ALTY	X015
16	u	75	U	55	NAK	15	ALTU	X016
17	i	69	I	49	HT	09	ALTI	X017
18	o	6F	O	4F	SI	0F	ALTO	X018
19	p	70	P	50	DLE	10	ALTP	X019
1A		5B	{	7B	ESC	1B	—	—
1B		5D	}	7D	GS	1D	—	—
1C	CR	0D	CR	0D	LF	0A	—	—
1D	CTRL	—	CTRL	—	CTRL	—	CTRL	—
1E	a	61	A	41	SOH	01	ALTA	X01E
1F	s	73	S	53	DC3	13	ALTS	X01F
20	d	64	D	44	EOT	04	ALTD	X020
21	f	66	F	46	ACK	06	ALTf	X021
22	g	67	G	47	BEL	07	ALTG	X022
23	h	68	H	48	BS	08	ALTh	X023
24	j	6A	J	4A	LF	0A	ALTJ	X024
25	k	6B	K	4B	VT	0B	ALTk	X025
26	l	6C	L	4C	FF	0C	ALTl	X026
27	;	3B	:	3A	—	—	—	—
28	'	27	"	22	—	—	—	—
29	`	60	-	7E	—	—	—	—
2A	left SHIFT	—	left SHIFT	—	left SHIFT	—	left SHIFT	—
2B	\	5C		7C	FS	1C	—	—
2C	z	7A	Z	5A	SUB	1A	ALTZ	X02C
2D	x	78	X	58	CAN	18	ALTx	X02D
2E	c	63	C	43	ETX	03	ALTC	X02E
2F	v	76	V	56	SYN	16	ALTv	X02F

key # - SCAN CODE	NORM CASE (ASCII code)		UPPER CASE (ASCII code)		CTRL CASE (ASCII code)		ALT CASE (ASCII code)	
30	b	62	B	42	STX	02	ALTB	X030
31	n	6E	N	4E	SO	0E	ALTN	X031
32	m	6D	M	4D	CR	0D	ALTM	X032
33	,	2C	<	3C	—	—	—	—
34	.	2E	>	3E	—	—	—	—
35	/	2F	?	3F	—	—	—	—
36	right SHIFT	—	right SHIFT	—	right SHIFT	—	right SHIFT	—
37	*	2A	PrScr**	—	CPrScr**	X072	—	—
38	ALT	—	ALT	—	ALT	—	ALT	—
39	SPACE	20	SPACE	20	SPACE	20	SPACE	X020
3A	CAPS	—	CAPS	—	—	—	CAPS	—
3B	F1	X03B	F11	X054	F21	X05E	F31	X068
3C	F2	X03C	F12	X055	F22	X05F	F32	X069
3D	F3	X03D	F13	X056	F23	X060	F33	X06A
3E	F4	X03E	F14	X057	F24	X061	F34	X06B
3F	F5	X03F	F15	X058	F25	X062	F35	X06C
40	F6	X040	F16	X059	F26	X063	F36	X06D
41	F7	X041	F17	X05A	F27	X064	F37	X06E
42	F8	X042	F18	X05B	F28	X065	F38	X06F
43	F9	X043	F19	X05C	F29	X066	F39	X070
44	F10	X044	F20	X05D	F30	X067	F40	X071
45	NUM LOCK	—	NUM LOCK	—	PAUSE **	—	NUM LOCK	—
46	SCROLL LOCK	—	SCROLL LOCK	—	BREAK **	—	SCROLL LOCK	—
54	SYS**	—	SYS**	—	SYS**	—	SYS**	—

Numeric key pad

SCAN CODE	NUM LOCK (ASCII code)		BASE CASE (ASCII code)		CTRL CASE (ASCII code)		ALT CASE (ASCII code)	
47	7	37	HOME	X047	CLR SCN	X077	†	—
48	8	38	↑	X048	—	—	†	—
49	9	39	PG UP	X049	TOP OF TEXT AND HOME	X084	†	—
4A	—	2D	—	2D	—	—	—	—
4B	4	34	*	X04B	LEFT ONE WORD	X073	†	—
4C	5	35	—	—	—	—	†	—
4D	6	36	*	X04D	RIGHT ONE WORD	X074	†	—
4E	+	2B	+	2B	—	—	—	—
4F	1	31	END	X04F	ERASE TO EOL	X075	†	—
50	2	32	↓	X050	—	—	†	—
51	3	33	PG DN	X051	ERASE TO EOS	X076	†	—
52	0	30	INS	X052	—	—	†	—
53	—	2E	DEL	X053	—	—	—	—

— indicates that no ASCII code is generated.

** indicates that the keys perform the special function noted.

† indicates that you can generate the character by holding down **ALT** while you type the decimal number on the keypad.

Characters Recognized by GW-BASIC

The GW-BASIC character set includes all characters that are legal in GW-BASIC commands, statements, functions, and variables. The set is made up of alphabetic, numeric, and special characters.

The alphabetic characters in GW-BASIC are the uppercase and lowercase letters of the alphabet.

The numeric characters in GW-BASIC are the digits 0 through 9.

The following special characters and terminal keys are recognized by GW-BASIC:

Character	Description
	Blank.
=	Equal sign or assignment symbol.
+	Plus sign or string concatenation.
-	Minus sign.
*	Asterisk or multiplication symbol.
/	Slash or division symbol.
^	Caret, exponentiation symbol.
(Left parenthesis.
)	Right parenthesis.
%	Percent or integer declaration.
#	Number sign or double-precision declaration.
\$	Dollar sign or string declaration.
!	Exclamation point or single-precision declaration.
[Left bracket.
]	Right bracket.
,	Comma.
"	Double quotation marks or string delimiter.
.	Period, dot, or decimal point.
'	Single quotation mark, apostrophe, or remark indicator.
;	Semicolon or carriage return suppressor.

Character	Description
:	Colon or line statement delimiter.
&	Ampersand or descriptor for hexadecimal and octal number conversion.
?	Question mark.
<	Less than symbol.
>	Greater than symbol.
\	Backslash or integer division symbol.
_	Underscore.
BACKSPACE	Deletes last character typed.
ESC	Erases the current logical line from the screen.
TAB	Moves print position to next tab stop. Tab stops are at every eight columns.
LINEFEED	Moves cursor to next physical line.
ENTER	Terminates input to a line and moves cursor to beginning of the next line, or executes statement in direct mode.

Glossary

abend An acronym for **ab**normal **end** of task. An abend is the termination of computer processing on a job or task prior to its completion because of an error condition that cannot be resolved by programmed recovery procedures.

access The process of seeking, reading, or writing data on a storage unit.

access methods Techniques and programs used to move data between main memory and input/output devices.

accuracy The degree of freedom from error. Accuracy is often confused with precision, which refers to the degree of preciseness of a measurement.

acronym A word formed by the initial letters of words or by initial letters plus parts of several words. Acronyms are widely used in computer technology. For example, COBOL is an acronym for COMmon Business-Oriented Language.

active partition A section of the computer's memory that houses the operating system being used.

address A name, label, or number identifying a register, location, or unit where information is stored.

algebraic language A language whose statements are structured to resemble the structure of algebraic expression. Fortran is a good example of an algebraic language.

algorithm A set of well-defined rules or procedures to be followed in order to obtain the solution of a problem in a finite number of steps. An algorithm can involve arithmetic, algebraic, logical, and other types of procedures and instructions. An algorithm can be simple or complex. However, all algorithms must produce a solution within a finite number

of steps. Algorithms are fundamental when using a computer to solve problems, because the computer must be supplied with a specific set of instructions that yields a solution in a reasonable length of time.

alphabetic Data representation by alphabetical characters in contrast to numerical; the letters of the alphabet.

alphanumeric A contraction of the words alphabetic and numeric; a set of characters including letters, numerals, and special symbols.

application The system or problem to which a computer is applied. Reference is often made to an application as being either of the computational type, in which arithmetic computations predominate, or of the data processing type, in which data handling operations predominate.

application program A computer program designed to meet specific user needs.

argument 1) A type of variable whose value is not a direct function of another variable. It can represent the location of a number in a mathematical operation or the number with which a function works to produce its results. 2) A known reference factor that is required to find a desired item (function) in a table. For example, in the square root function $\text{SQRT}(X)$, X is the argument. The value of X determines the square root value returned by this function.

array 1) An organized collection of data in which the argument is positioned before the function. 2) A group of items or elements in which the position of each item or element is significant. A multiplication table is a good example of an array.

ASCII Acronym for American Standard Code for Information Interchange. ASCII is a standardized, eight-bit code used by most computers for interfacing. ASCII was developed by the American National Standards Institute (ANSI). It uses seven binary bits for information and the eighth bit for parity purposes.

assembler A computer program that produces a machine language program, which can then be directly executed by the computer.

assembly language A symbolic language that is machine-oriented rather than problem-oriented. A program in an assembly language is converted by an assembler to a machine language program. Symbols representing storage locations are converted to numerical storage locations; symbolic operation codes are converted to numeric operation codes.

asynchronous 1) Activities that do not have a regular time or clock relationship. *See* synchronous. 2) A type of computer operation in which a new instruction is initiated when the former instruction is completed. Thus, there is no regular time schedule, or clock, with respect to instruction sequence. The current instruction must be complete before the next is begun, regardless of the length of time the current instruction takes to reach completion.

asynchronous communication A way of transmitting data serially from one device to another, in which each transmitted character is preceded by a start bit and followed by a stop bit. This is also called start/stop transmission.

backup 1) A second copy of data on a diskette or other medium, ensuring recovery from loss or destruction of the original media 2) On-site or remote equipment available to complete an operation in the event of primary equipment failure.

BASIC Acronym for Beginner's All-purpose Symbolic Instruction Code. BASIC is a computer programming language developed at Dartmouth College as an instructional tool in teaching fundamental programming concepts. This language has since gained wide acceptance as a time-sharing language and is considered one of the easiest programming languages to learn.

batch processing A method of operating a computer so that a single program or set of related programs must be completed before the next type of program is begun.

baud A unit of measurement of data processing speed. The speed in bauds is the number of signal elements per second. Because a signal element can represent more than one bit, *baud* is not synonymous with *bits per second*. Typical baud rates are 110, 300, 1200, 2400, 4800, and 9600.

binary 1) A characteristic or property involving a choice or condition in which there are two possibilities. 2) A numbering system that uses 2 as its base instead of 10 (as in the decimal system). The binary system uses only two digits, 0 and 1, in its written form. 3) A device whose design uses only two possible states or levels to perform its functions. A computer executes programs in binary form.

binary digit A quantity that is expressed in the binary digits of 0 and 1.

bit A contraction of *binary digit*. A bit can either be 0 or 1, and is the smallest unit of information recognizable by a computer.

block An amount of storage space or data, of arbitrary length, usually contiguous, and often composed of several similar records, all of which are handled as a unit.

Boolean logic A field of mathematical analysis in which comparisons are made. A programmed instruction can cause a comparison of two fields of data and modify one of those fields or another field as a result of comparison. This system was formulated by British mathematician George Boole (1815-1864). Some Boolean operators are OR, AND, NOT, XOR, EQV, and IMP.

boot A machine procedure that allows a system to begin operations at the desired level by means of its own initiation. The first few instructions are loaded into a computer from an input device. These instructions allow the rest of the system to be loaded. The word *boot* is abbreviated from the word *bootstrap*.

bps Abbreviation for *bits per second*.

buffer A temporary storage area from which data is transferred to or from various devices.

built-in clock A real-time clock that lets your programs use the time of day and date. Built into MS-DOS, it lets you set the timing of a program. It can be used to keep a personal calendar, and it automatically measures elapsed time.

byte An element of data composed of eight data bits plus a parity bit and representing either one alphabetic or special character, two decimal digits, or eight binary bits. *Byte* is also used to refer to a sequence of eight binary digits handled as a unit. It is usually encoded in the ASCII format.

calculation A series of numbers and mathematical signs that, when entered into a computer, is executed according to a series of instructions.

central processor (CPU) The heart of the computer system, where data is manipulated and calculations are performed. The CPU contains a control unit to interpret and execute the program and an arithmetic-logic unit to perform computations and six logical processes. It also routes information, controls input and output, and temporarily stores data.

chaining The use of a pointer in a record to indicate the address of another record logically related to the first.

character Any single letter of the alphabet, numeral, punctuation mark, or other symbol that a computer can read, write, and store. *Character* is synonymous with the term *byte*.

COBOL Acronym for COmmon Business-Oriented Language, a computer language suitable for writing complicated business applications programs. It was developed by CODASYL, a committee representing the U. S. Department of Defense, certain computer manufacturers, and major users of data processing equipment. COBOL is designed to express data manipulations and processing problems in English narrative form, in a precise and standard manner.

code 1) To write instructions for a computer. 2) To classify data according to arbitrary tables. 3) To use a machine language. 4) To program.

command A pulse, signal, word, or series of letters that tells a computer to start, stop, or continue an operation in an instruction. *Command* is often used incorrectly as a synonym for *instruction*.

compatible A description of data, programs, or equipment that can be used between different kinds of computers or equipment.

compiler A computer program that translates a program written in a problem-oriented language into a program of instructions similar to, or in, the language of the computer.

computer network A geographically dispersed configuration of computer equipment connected by communication lines and capable of load sharing, distributive processing, and automatic communication between the computers within the network.

concatenate To join together data sets, such as files, in a series to form one data set, such as one new file. The term concatenate literally means *to link together*. A concatenated data set is a collection of logically connected data sets.

configuration In hardware, a group of interrelated devices that constitute a system. In software, the total of the software modules and their interrelationships.

constant A never-changing value or data item.

coprocessor A microprocessor device connected to a central microprocessor that performs specialized computations (such as floating-point arithmetic) much more efficiently than the CPU alone.

cursor A blinking line or box on a computer screen that indicates the next location for data entry.

data A general term used to signify all the basic information elements that can be produced or processed by a computer. *See* information.

data element The smallest named physical data unit.

data file A collection of related data records organized in a specific manner. Data files contain computer records that hold information, as opposed to holding data handling information or a program.

debug The process of checking the logic of a computer program to isolate and remove mistakes from the program or other software.

default An action or value that the computer automatically assumes unless a different instruction or value is given.

delimit To establish parameters; to set a minimum and a maximum.

delimiter A character that marks the beginning or end of a unit of data on a storage medium. Commas, semicolons, periods, and spaces are used as delimiters to separate and organize items of data.

detail file A data file composed of records having similar characteristics but containing data which is relatively changeable by nature, such as employee weekly payroll data. *See also* master file.

device A piece of hardware that can perform a specific function. A printer is an example of a device.

diagnostic programs Special programs used to align equipment or isolate equipment malfunctions.

directory A table that gives the name, location, size, and the creation or last revision date for each file on the storage media.

diskette A flat, flexible platter coated with magnetic material, enclosed in a protective envelope, and used for storage of software and data.

disk operating system A collection of procedures and techniques that enable the computer to operate using a disk drive system for data entry and storage. Usually abbreviated as DOS.

DOS Acronym for *disk operating system*.

double-density A type of diskette that has twice the storage capacity of standard single-density diskettes.

double-precision The use of two computer words to represent each number. This technique allows the use of twice as many digits as are normally available and is used when extra precision is needed in calculations.

double-sided A term that refers to a diskette that can contain data on both its surfaces.

drive A device that holds and manipulates magnetic media so that the CPU can read data from or write data to the media.

end-of-file mark (EOF) A symbol or machine equivalent that indicates that the last record of a file has been read.

erase To remove or replace magnetized spots from a storage medium.

error message An audible or visual indication of hardware or software malfunction or of an illegal data-entry attempt.

execute To carry out an instruction or perform a routine.

exponent A symbol written above a factor and on the right, telling the number of times the factor is repeated. In the example of A^2 , A is the factor and 2 is the exponent. A^2 means A times A ($A \times A$).

extension A set of 1-3 characters that follows a filename. The extension further defines or clarifies the filename. It is separated from the filename by a period (.).

field An area of a record that is allocated for a specific category of data.

file A collection of related data or programs that is treated as a unit by the computer.

file protection The devices or procedures that prevent unintentional erasure of data on a storage device such as a diskette.

file structure A conceptual representation of the way data values, records, and files are related to each other. The structure usually implies how the data is stored and how it must be processed.

filename The unique name, usually assigned by a user, that identifies one file for all subsequent operations that use that file.

fixed disk A hard disk enclosed in a permanently sealed housing that protects it from environmental interference. Used for data storage.

floating-point arithmetic A method of calculation in which the computer or program automatically records and accounts for the location of the radix point. The programmer need not consider the radix location.

floating-point routine A set of program instructions that permits a floating-point mathematics operation in a computer lacking the feature of automatically accounting for the radix point.

format A predetermined arrangement of data that structures the storage of information on an external storage device.

function A computer action, as defined by a specific instruction. Some GW-BASIC functions are COS, EOF, INSTR, LEFT\$, and TAN.

function keys Specific keys on the keyboard that, when pressed, instruct the computer to perform a particular operation. The function of the keys is determined by the applications program being used.

GIGO Acronym for Garbage In, Garbage Out. An informal term that indicates sloppy data processing. Normally used to make the point that if the input data is bad (garbage in) then the output data will also be bad (garbage out).

global search Used in reference to a variable (character or command), it causes the computer to locate all occurrences of that variable.

graphics A hardware/software ability to display objects in pictures, rather than words, usually on a graphic (CRT) display terminal with line-drawing capability and permitting interaction, such as the use of a light pen.

hard copy A printed copy of computer output in a readable form, such as reports, checks, or plotted graphs.

hardware The physical equipment that makes up a system.

hexadecimal A number system with a base, or radix, of 16. The symbols used in this system are the decimal digits 0-9 and six additional digits generally represented as A, B, C, D, E, and F.

hidden files Files that cannot be seen during normal directory searches.

hierarchical directories See tree-structured directories.

housekeeping functions Routine operations that must be performed before the actual processing begins or after it is complete.

information Facts and knowledge derived from data. The computer operates on and generates data. The meaning derived from the data is information. That is, information results from data; the two words are not synonymous, although they are often used interchangeably.

interpreter A program that reads, translates, and executes a user's program, such as one written in the BASIC language, one line at a time. A compiler, on the other hand, reads and translates the entire user's program before executing it.

input 1) The process or device involved in sending data to a computer.
2) Actual data being entered into a computer.

input/output A general term for devices that communicate with a computer. Input/output is usually abbreviated as I/O.

instruction A program step that tells the computer what to do next. *Instruction* is often used incorrectly as a synonym for *command*.

integer A complete entity, having no fractional part. The whole or natural number. For example, 65 is an integer; 65.1 is not.

integrated circuit A complete electronic circuit contained in a small semiconductor component.

interface An information interchange path that allows parts of a computer, computers and external equipment (such as printers, monitors, or modems) or two or more computers to communicate or interact.

I/O Acronym for *input/output*.

job A collection of tasks viewed by the computer as a unit.

K The symbol signifying the quantity 210, which is equal to 1024. K is sometimes confused with the symbol k, (kilo) which is equal to 1000.

logarithm A *logarithm* of a given number is the value of the exponent indicating the power required to raise a specified constant, known as the *base*, to produce that given number. That is, if B is the base, N is the given number, and L is the logarithm, then $BL = N$. Since $10^3 = 1000$, the logarithm to the base 10 of 1000 is 3.

loop A series of computer instructions that are executed repeatedly until a desired result is obtained or a predetermined condition is met. The ability to loop and reuse instructions eliminates repetitious instructions and is one of the most important attributes of stored programs.

M The symbol signifying the quantity 1,000,000 (106). When used to denote storage, it more precisely refers to 1,048,576 (220).

mantissa The fractional or decimal part of a logarithm of a number. For example, the logarithm of 163 is 2.212. The mantissa is 0.212, and the characteristic is 2.0. In floating-point numbers, the mantissa is the number part. For example, the number 24 can be written as 24,2 where 24 is the mantissa and 2 is the exponent. The floating-point number is read as .24 X 10², or 24.

master file A data file composed of records having similar characteristics that rarely change. A good example of a master file would be an employee name and address file that also contains social security numbers and hiring dates.

media The plural of *medium*.

medium The physical material on which data is recorded and stored. Magnetic tape, punched cards, and diskettes are examples of media.

memory The high-speed work area in the computer where data can be held, copied, and retrieved.

menu A list of choices from which an operator can select a task or operation to be performed by the computer.

microprocessor A semiconductor central processing unit (CPU) in a computer.

modem Acronym for *modulator demodulator*. A modem converts data from a computer to analog signals that can be transmitted through telephone lines, or converts the signals from telephone lines into a form the computer can use.

MS-DOS Acronym for Microsoft Disk Operating System.

nested programs or subroutines A program or subroutine that is incorporated into a larger routine to permit ready execution or access of each level of the routine. For example, nesting loops involves incorporating one loop of instructions into another loop.

null Empty or having no members. This is in contrast to a blank or zero, which indicates the presence of no information. For example, in the number 540, zero contains needed information.

numeric A reference to numerals as opposed to letters or other symbols.

octal number system A representation of values or quantities with octal numbers. The octal number system uses eight digits: 0, 1, 2, 3, 4, 5, 6, and 7, with each position in an octal numeral representing a power of 8. The octal system is used in computing as a simple means of expressing binary quantities.

operand A quantity or data item involved in an operation. An operand is usually designated by the address portion of an instruction, but it can also be a result, a parameter, or an indication of the name or location of the next instruction to be executed.

operating system An organized group of computer instructions that manages the overall operation of the computer.

operator A symbol indicating an operation and itself the subject of the operation. It indicates the process that is being performed. For example, + is addition, - is subtraction, X is multiplication, and / is division.

option. An add-on device that expands a system's capabilities.

output Computer results, or data that has been processed.

parallel output The method by which all bits of a binary word are transmitted simultaneously.

parameter A variable that is given a value for a specific program or run. A definable characteristic of an item, device, or system.

parity An extra-bit of code that is used to detect data errors in memory by making the sum of the active bit in a data word either an odd or an even number.

partition An area on a fixed disk set aside for a specific purpose, such as a location for an operating system.

peripheral An external input/output or storage device.

pixel Acronym for *picture element*. A pixel is a single dot on a monitor that can be addressed by a single bit.

port The entry channel to and from the central computer for connection of a communications line or other peripheral device.

power The functional area of a system that transforms an external power source into internal DC supply voltage.

program A series of instructions or statements in a form acceptable to a computer, designed to cause the computer to execute a series of operations. Computer programs include software such as operating systems, assemblers, compilers, interpreters, data management systems, utility programs, sort-merge programs, and maintenance/diagnostic programs, as well as application programs such as payroll, inventory control, and engineering analysis programs.

prompt A character or series of characters that appears on the screen to request input from the user.

RAM Acronym for *random-access memory*.

radian The natural unit of measure of the angle between two intersecting half-lines on the angles from one half-line to another intersecting half-line. It is the angle subtended by an arc of a circle equal in length to the radius of the circle. As the circumference of a circle is equal to 2π times its radius, the number of radians in an angle of 360° or in a complete turn is 2π .

radix A number that is arbitrarily made the fundamental number of a system of numbers; a *base*. Thus, 10 is the radix, or base, of the common system of logarithms, and also of the decimal system of enumeration.

random-access memory The system's high-speed work area that provides access to memory storage locations by using a system of vertical and horizontal coordinates. The computer can write information into or read information from the random-access memory. Random-access memory is often called RAM.

raster unit On a graphic display screen, a *raster unit* is the horizontal or vertical distance between two adjacent addressable points on the screen.

read-only memory A type of memory that contains permanent data or instructions. The computer can read from but not write to the read-only memory. Read-only memory is often called ROM.

real numbers An ordinary number, either rational or irrational; a number in which there is no imaginary part; a number generated from the single unit, 1; any point in a continuum of natural numbers filled in with all rationals and all irrationals and extended indefinitely, both positive and negative.

real time 1) The actual time required to solve a problem. 2) The process of solving a problem during the actual time that a related physical process takes place so that results can be used to guide the physical process.

remote A term used to refer to devices that are located at sites away from the central computer.

reverse video A display of characters on a background, opposite of the usual display.

ROM Acronym for *read-only memory*.

RS-232 A standard communications interface between a modem and terminal devices that complies with EIA Standard RS-232.

serial output Sending only one bit at a time to and from interconnected devices.

single-density The standard recording density of a diskette. Single-density diskettes can store approximately 3400 bits per inch (bpi).

single-precision value The number of words or storage positions used to denote a number in a computer. Single-precision arithmetic is the use of one word per number, double-precision arithmetic is the use of two words per number, and so on. For variable-word-length computers, *precision* is the number of digits used to denote a number. The higher the precision, the greater the number of decimal places that can be carried.

single-sided A term used to describe a diskette that contains data on one side only.

software A string of instructions that, when executed, direct the computer to perform certain functions.

stack architecture An architecture wherein any portion of the external memory can be used as a last-in, first-out stack to store/retrieve the contents of the accumulator, the flags, or any of the data registers. Many units contain a 16-bit stack pointer to control the addressing of this external stack. One of the major advantages of the stack is that multiple-level interrupts can be handled easily, since complete system status can be saved when an interrupt occurs and then be restored after the interrupt. Another major advantage is that almost unlimited subroutine nesting is possible.

statement A high-level language instruction to the computer to perform some sequence of operations.

synchronous A type of computer operation in which the execution of each instruction or each event is controlled by a clock signal: evenly spaced pulses that enable the logic gates for the execution of each logic step. A synchronous operation can cause time delays by causing waiting for clock signals, although all other signals at a particular logic gate were available. *See* asynchronous.

switch. An instruction, added to a command, that designates a course of action, other than default, for the command process to follow.

syntax Rules of statement structure in a programming language.

system A collection of hardware, software, and firmware that is interconnected to operate as a unit.

task A machine run; a program in execution.

toggle Alternation of function between two stable states.

track A specific area on a moving-storage medium, such as a diskette, disk, or tape cartridge, that can be accessed by the drive heads.

tree-structured directory A file organization structure, consisting of directories and subdirectories that, when diagrammed, resembles a tree.

truncation To end a computation according to a specified rule; for example, to drop numbers at the end of a line instead of rounding them off, or to drop characters at the end of a line when a file is copied.

upgrade To expand a system by installing options or using revised software.

utility function Computer programs, dedicated to one particular task, that help you use your computer. For example, FDISK is a utility function for setting up partitions on the fixed disk.

variable A quantity that can assume any of a set of values as a result of processing data.

volume label The name for the contents of a diskette or a partition on a fixed disk.

word The largest unit of bits that the computer can handle in a single operation.

write-protect notch A cut-out opening in the sealed envelope of a diskette that, when covered, prevents writing or adding text to the diskette, but allows information to be read from the diskette.

Tandy
GW-BASIC
User's Reference

Contents

Introduction	1
ABS Function	2
ASC Function	3
ATN Function	4
AUTO Command	5
BEEP Statement	7
BLOAD Command	8
BSAVE Command	10
CALL Statement	11
CDBL Function	15
CHAIN Statement	16
CHDIR Command	18
CHR\$ Function	19
CINT Function	20
CIRCLE Statement	21
CLEAR Command	23
CLOSE Statement	25
CLS Statement	26
COLOR Statement	28
COM(n) Statement	31
COMMON Statement	32
CONT Command	33
COS Function	34
CSNG Function	35
CSRLIN Variable	36
CVI, CVS, CVD Functions	37
DATA Statements	39
DATE\$ Statement and Variable	41
DEF FN Statement	43
DEFINT/SNG/DBL/STR Statements	45
DEF SEG Statement	47
DEF USR Statement	49
DELETE Command	51

DIM Statement	52
DRAW Statement	53
EDIT Command	57
END Statement	58
ENVIRON Statement	59
ENVIRON\$ Function	61
EOF Function	64
ERASE Statement	65
ERDEV and ERDEV\$ Variables	66
ERR and ERL Variables	67
ERROR Statement	68
EXP Function	70
EXTERR Function	71
FIELD Statement	72
FILES Command	73
FIX Function	75
FOR and NEXT Statements	76
FREE Function	79
GET Statement	80
GET Statement (Graphics)	81
GOSUB...RETURN Statement	83
GOTO Statement	85
HEX\$ Function	86
IF Statement	87
INKEY\$ Variable	89
INP Function	91
INPUT Statement	92
INPUT# Statement	95
INPUT\$ Function	96
INSTR Function	98
INT Function	100
IOCTL Statement	101
IOCTL\$ Function	102
KEY Statement	103
KEY(number) Statement	106
KILL Command	107
LEFT\$ Function	108
LEN Function	109

LET Statement	110
LINE Statement	111
LINE INPUT Statement	115
LINE INPUT# Statement	117
LIST Command	119
LLIST Command	121
LOAD Command	122
LOC Function	123
LOCATE Statement	124
LOC Statement	126
LOF Function	128
LOG Function	129
LPOS Function	130
LPRINT and LPRINT USING Statements	131
LSET Statement	132
MERGE Command	133
MID\$ Function	134
MID\$ Statement	135
MKDIR Command	136
MKI\$, MKS\$, MKD\$ Functions	137
NAME Command	138
NEW Command	139
OCT\$ Function	140
ON Statement	141
ON ERROR GOTO Statement	147
ON/GOSUB and ON/GOTO Statements	149
OPEN STATEMENT	150
OPEN "COM Statement	156
OPTION BASE Statement	159
OUT Statement	160
PAINT Statement	161
PALETTE, PALETTE USING Statements	165
PCOPY Command	169
PEEK Command	170
PEN Function	171
PEN Statement	173
PLAY Statement	174
PLAY Function	177

PMAP Function (Graphics)	178
POINT Function	179
POKE Statement	181
POS Function	182
PSET Statement	183
PRINT Statement	185
PRINT USING Statement	187
PRINT# and PRINT# USING Statements	192
PRESET Statement	195
PUT Statement (Files)	196
PUT Statement (Graphics)	197
RANDOMIZE Statement	200
READ Statement	202
REM Statement	204
RENUM Command	206
RESET Command	208
RESTORE Statement	209
RESUME Statement	210
RETURN Statement	211
RIGHT\$ Function	212
RMDIR Command	213
RND Function	214
RSET Statement	215
RUN Command	216
SAVE Command	217
SCREEN Function	218
SCREEN Statement	219
SGN Function	226
SHELL Statement	227
SIN Function	229
SOUND Statement	230
SPACE\$ Function	233
SPC Function	234
SQR Function	235
STICK Function	236
STOP Statement	237
STR\$ Function	238
STRING Statement and Function	239

STRIG(number) Statement	241
STRING\$ Function	242
SWAP Statement	243
SYSTEM Command	244
TAB Function	245
TAN Function	246
TIMES\$ Statement and Variable	247
TIMER Function	249
TRON/TROFF Commands	250
UNLOCK Statement	251
USR Function	254
VAL Function	256
VARPTR Function	257
VARPTR\$ Function	260
VIEW Statement	261
VIEW PRINT Statement	263
WAIT Statement	264
WHILE-WEND Statement	265
WIDTH Statement	267
WINDOW Statement	269
WRITE Statement	272
WRITE# Statement	273

Tandy GW-BASIC User's Reference

Introduction

This manual provides an alphabetical reference to all of GW-BASIC's *keywords* (statements, functions, commands, and variables).

The name and type of each instruction appears at the top of the page, followed by these subheadings:

- Purpose:** A statement telling the purpose of the instruction.
- Syntax:** The complete notation of the instruction, using *option names* to represent actual data and values you might type. These option names appear in italics and are explained in the "Comments:" section.
- Comments:** Additional information about the instruction, and how GW-BASIC responds to it.
- Examples:** An illustration of the instruction as it might appear in a program. In this case, example lines use real values and data rather than option names.

For more information on using the references in this manual, see the *Tandy GW-BASIC User's Guide*.

ABS Function

Purpose:

To return the absolute value of the expression *number*.

Syntax:

ABS(number)

Comments:

number must be a numeric expression.

Examples:

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

This program line returns 35 as the result of the action.

ASC Function

Purpose:

To return a numeric value that is the ASCII code for the first character of the specified *string*.

Syntax:

ASC(*string*)

Comments:

If *string* is null, GW-BASIC returns an Illegal Function Call error.

If *string* begins with an uppercase letter, GW-BASIC returns a value in the range 65-90.

If *string* begins with a lowercase letter, GW-BASIC returns a value in the range 97-122.

If *string* begins with 0-9, GW-BASIC returns a value in the range 48-57.

See the CHR\$ function for ASCII-to-string conversion.

See Appendix B in the *Tandy GW-BASIC User's Guide* for ASCII codes.

Examples:

```
10 STRING = "TEN"  
20 PRINT ASC(STRING)
```

These program lines return 84. The value 84 is the ASCII code for the letter T.

ATN Function

Purpose:

To return the arctangent of *number*, when *number* is expressed in radians.

Syntax:

ATN(*number*)

Comments:

The result is in the range $-\pi/2$ to $\pi/2$.

The expression *number* can be any numeric type. GW-BASIC performs the evaluation of ATN in single precision unless you execute GW-BASIC using the /d switch.

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

Examples:

```
10 INPUT NUMBER
20 PRINT ATN(NUMBER)
```

If you run the program and enter 3 as NUMBER, this example prints the arctangent of 3 radians (1.249046).

AUTO Command

Purpose:

To generate and increment line numbers automatically each time you press **ENTER**.

Syntax:

```
AUTO [line number][,[increment]]  
AUTO . [[increment]]
```

Comments:

AUTO is useful for program entry because it makes typing line numbers unnecessary.

AUTO begins numbering at *line number* and increments each subsequent line number by the user-specified *increment*. The default for both values is 10.

You can use the period (.) as a substitute for *line number* to indicate the current line.

If you type a comma after *line number* but do not specify an *increment*, GW-BASIC uses the last increment specified in an AUTO command.

If AUTO generates a line number that is already in use, an asterisk appears after the number to warn that any input replaces the existing line. However, pressing **ENTER** immediately after the asterisk saves the line and generates the next line number.

Enter **CTRL** **BREAK** or **CTRL** **C** to terminate AUTO and cause GW-BASIC to return to command level.

Note: GW-BASIC does not save the line you are editing when you press **CTRL** **BREAK** or **CTRL** **C**.

Examples:

`AUTO 100,50`

Generates line numbers 100, 150, 200, and so on.

`AUTO`

Generates line numbers 10, 20, 30, 40, and so on.

BEEP Statement

Purpose:

To sound the speaker at 800 Hz (800 cycles per second) for one-quarter of a second.

Syntax:

BEEP

Comments:

BEEP, `CTRL G`, and `PRINT CHR$(7)` have the same effect.

Examples:

```
2340 IF COUNT > 20 THEN BEEP
```

If count is out of range (greater than 20), the computer beeps.

BLOAD Command

Purpose:

To load an image file anywhere in user memory.

Syntax:

BLOAD *pathname* [*offset*]

Comments:

pathname is a valid string expression containing the device and filename.

offset is a valid numeric expression in the range 0-65535. This is the offset into the segment, declared by the last DEF SEG statement, where loading is to start.

If you omit *offset*, the offset specified at BSAVE is assumed; that is, the file is loaded into the same location it was saved from.

Note: BLOAD does not perform an address range check. Therefore, it is possible to accidentally BLOAD anywhere in memory.

You must not BLOAD over the GW-BASIC stack space, a GW-BASIC program, or the GW-BASIC variable area.

While BLOAD and BSAVE are useful for loading and saving machine language programs, they are not restricted to them. The DEF SEG statement lets you specify any segment as the source or target for BLOAD and BSAVE. For example, this allows the video screen buffer to be read from or written to the diskette. BLOAD and BSAVE are useful in saving and displaying graphic images.

Examples:

```
10 DEF SEG = &HB800
20 BLOAD "PICTURE", 0
```

The DEF SEG statement in Line 10 points the segment at the screen buffer.

The DEF SEG statement in Line 10 and the offset of 0 in Line 20 guarantee that the correct address is used.

The BLOAD command in Line 20 loads the file named Picture into the screen buffer.

Note: The BSAVE example in the next section illustrates how the file named Picture is saved.

BSAVE Command

Purpose:

To save portions of user memory on the specified device.

Syntax:

BSAVE *pathname*, *offset*, *length*

Comments:

pathname is a valid string expression containing the filename.

offset is a valid numeric expression in the range 0-65535. This is the offset into the segment, declared by the last DEF SEG, where saving is to start.

length is a valid numeric expression in the range 0-65535, specifying the length of the memory image to be saved.

If *pathname* is less than one character, GW-BASIC issues a Bad File Number error and terminates the load operation.

Execute a DEF SEG statement before the BSAVE. The last known DEF SEG address is always used for the save.

You must use the DEF SEG statement to set up the segment address to the start of the screen buffer. An offset of 0 and a length of 16384 tell BSAVE to save the entire 16K screen buffer.

Examples:

```
10 DEF SEG = &HB800
20 BSAVE"PICTURE",0,16384
```

In Line 10, the DEF SEG statement points the segment at the screen buffer.

The BSAVE command in Line 20 saves the screen buffer in the file named Picture.

CALL Statement

Purpose:

To call an assembly (or machine) language subroutine.

Syntax:

`CALL numvar [(variables)]`

Comments:

numvar is the starting point in memory of the subroutine being called as an offset into the current segment.

variables are the variables or constants, separated by commas and enclosed in parentheses, that CALL is to pass to the routine.

Use CALL to interface with assembly language programs unless you must retain compatibility with previous version of GW-BASIC. Although you can use `USR`, CALL is compatible with more languages, produces a more readable source code, and can pass multiple arguments.

When GW-BASIC encounters the CALL statement it:

- Pushes each parameter location in the variable onto the stack. The parameter location is a 2-byte offset into GW-BASIC's data segment.
- It pushes the return address code segment (CS) and the offset onto the stack.
- It uses the segment address given in the last DEF SEG statement and the offset given in the variable name to transfer control to the user routine.

The user routine now has control. You can reference parameters by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP.

The called routine might destroy the contents of any registers except the segment registers.

The called program must know how many parameters were passed. Parameters are referenced by adding a positive offset to BP, assuming the called routine moved the current stack pointer into BP (that is, `MOV BP,SP`).

The called program must know the variable type for numeric parameters passed.

The called routine must do a RET *number*, where *number* is the number of parameters in the variable times 2. This is necessary in order to adjust the stack to the point at the start of the calling sequence.

The system returns values to GW-BASIC by including in the argument list the name of the variable that is to receive the result.

If the argument is a string, the parameter offset points to three bytes 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 eight bits of the string starting address in the string space.

If the argument is a string literal in the program, the string descriptor points to program text. Be careful not to alter or destroy a program this way. To avoid unpredictable results, add +"" to the string literal in the program, as in

```
20 A$="BASIC"+""
```

This forces the string literal to be copied into the string space. Now GW-BASIC can modified modify the string without affecting the program.

Note: You can alter strings with user routines but you must not change their length. GW-BASIC cannot correctly erase strings if their lengths are modified by external routines.

For more information on the CALL statement and USR function, see Appendix C in the *Tandy GW-BASIC User's Guide*.

Examples:

```
100 DEF SEG = &H2000
110 ARK = 0
120 CALL ARK(A,B$,C)
```

```
.
.
.
```

In this example, Line 100 sets the segment to hex 2000. The program sets ARK to zero so that the call to ARK executes the subroutine at location &H2000.

The following sequence of 8086 Assembly Language demonstrates access of the parameters passed and stored in variable C:

```
PUSH BP
MOV BP,SP           ; Gets current stack position in BP.
MOV BX,8[BP]        ; Gets address of B$ descriptor.
MOV CL,[BX]         ; Gets length of B$ in CL.
MOV DX,1[BX]        ; Gets address of B$ text in DX.
.
.
.
MOV SI,10[BP]       ; Gets address of A in SI.
MOV DI,6[BP]        ; Gets pointer to C in DI.
MOVSW              ; Stores variable A in C.
RET 6               ; Restores stack and returns.
```

MOVSW copies only two bytes. This is sufficient if Variables A and C are integer. The program must copy four bytes if they are single precision or eight bytes if they are double precision.

```
100 DEF EG = &H2000
110 CC = &H7FA
120 CALL ACC(A,B$,C)
```

```
.
.
.
```

In this previous example, Line 100 sets the segment to hex 2000. The example adds the value of variable ACC into the address as the low word after the DEF SEG value is shifted four bits to the left. (This is a function of the microprocessor, not of GW-BASIC.) The program sets ACC to &H7FA, so that the call to ACC executes the subroutine at the location hex 2000:7FA (absolute address hex 207FA).

CDBL Function

Purpose:

To convert *number* to a double-precision number.

Syntax:

`CDBL(number)`

Comments:

number must be a numeric expression.

Examples:

```
10 A = 454.67
20 PRINT A;CDBL(A)
```

These program lines returns two values, 454.67 and 454.6700134277344. The value 454.6700134277344 is the double-precision version of the single-precision value (454.67) stored in the variable named A.

The last 11 numbers in the double-precision number have no meaning in this example, since A was previously defined to only two-decimal place accuracy.

Note: See the CINT and CSNG functions for converting numbers to integer and single precision, respectively.

CHAIN Statement

Purpose:

To transfer control to the specified program and pass chain variables to it from the current program.

Syntax:

```
CHAIN [MERGE] pathname [, [line ][, [ALL][, DELETE range ]]]
```

Comments:

MERGE overlays the current program with the called program.

Note: The called program must be an ASCII file (previously saved with the a option) if it is to be merged. See the MERGE command.

pathname is the name of the program called to be chained to. GW-BASIC assumes the .bas extension unless you specify another.

line is a line number or an expression that corresponds to a line number in the called program. It is the starting point for execution of the called program. For example, the following begins execution of PROG1 at Line 1000:

```
10 CHAIN "PROG1",1000
```

If you omit *line*, execution begins at the first line.

line is not affected by a RENUM command. However, the line numbers in the specified range are affected by a RENUM command.

ALL specifies that every variable in the current program is chained to the called program. For example:

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

If you omit ALL, the current program must contain a COMMON statement to list the variables that are passed.

CHAIN executes a RESTORE before it runs the program that it is to be chained to. The READ statement then gets the first item in the DATA statement. Reading does not resume where it left off in the program that is being chained.

After GW-BASIC executes an overlay and no longer needs it, use the **DELETE** command to remove the overlay in order to bring in a new overlay.

The **CHAIN** statement with the **MERGE** command leaves the files open and preserves the current option base setting.

If you omit the **MERGE** command, the **OPTION BASE** setting is preserved and **CHAIN** preserves no variable types or user-defined functions for use by the chained program. That is, any **DEFINT**, **DEFSNG**, **DEFDBL**, **DEFSTR**, or **DEF FN** statement containing shared variables must be restated in the chained program.

When using the **MERGE** command, place user-defined functions before any **CHAIN MERGE** statements in the program. Otherwise, they are undefined after the merge is complete.

CHDIR Command

Purpose:

To change from one working directory to another.

Syntax:

`CHDIR pathname`

Comments:

pathname is a string expression of up to 63 characters.

To make Sales the working directory on Drive A and Inventory the working directory on Drive B (assume A is the default drive), type the following commands:

`CHDIR "SALES"`

`CHDIR "B:INVENTORY"`

CHR\$ Function

Purpose:

To convert an ASCII code to its equivalent character.

Syntax:

`CHR$(number)`

Comments:

number is a value in the range 0-255.

Normally, you use CHR\$ to send a special character to the terminal or printer. For example, you could send CHR\$(7) to sound a beep through the speaker as a preface to an error message, or you could send a form feed, CHR\$(12), to the printer.

See the ASC function for ASCII-to-numeric conversion.

ASCII codes are listed in Appendix B of the *Tandy GW-BASIC User's Guide*.

Examples:

```
PRINT CHR$(66);
```

Prints the ASCII character code 66, which is the uppercase letter B.

```
PRINT CHR$(13);
```

Prints a carriage return.

CINT Function

Purpose:

To round numbers with fractional portions to the next whole number or integer.

Syntax:

CINT(*number*)

Comments:

If *number* is not in the range -32768-32767, an Overflow error occurs.

See the FIX and INT functions, both of which return integers.

Examples:

```
PRINT CINT(45.67)
```

Returns 46. CINT rounds 45.67 up to 46.

Note: See the CDBL and CSNG functions for converting numbers to the double-precision and single-precision data types, respectively.

CIRCLE Statement

Purpose:

To draw a circles, ellipses, and angles on the screen in the graphics mode.

Syntax:

`CIRCLE(xcenter, ycenter), radius[[color][,start][,end][, aspect]]]`

Comments:

xcenter and *ycenter* are the x and y coordinates of the center of the ellipse, and *radius* is the radius (measured along the major axis) of the ellipse. The quantities *xcenter* and *ycenter* can be expressions. The center attributes can use either absolute or relative coordinates.

color specifies the ellipse color. Its value depends on the screen mode. In the medium-resolution mode, the COLOR statement defines which colors are selected. The ellipse is the same color as the background when the program selects 0. The default for the medium-resolution mode is 3. See the COLOR statement to learn to select foreground and background colors in graphics mode.

In the high-resolution mode, 0 indicates black and 1 indicates white. The default for the high resolution mode is 1.

The *start* and *end* angle parameters are radian arguments between -2π and 2π which specify where the drawing of the ellipse is to begin and end. If *start* or *end* is negative, the ellipse is connected to the center point with a line, and the angles are treated as if they are positive. (Note that this is different from adding 2π .)

aspect describes the ratio of the x radius to the y radius (x : y). The default aspect ratio is 5:6 in medium resolution, and 5:12 in high resolution. This gives a visual circle in either graphics mode, assuming a standard monitor screen aspect ratio of 4:3. If the aspect ratio is less than 1, then the radius is given in x-pixels. If it is greater than 1, the radius is given in y-pixels.

In many cases, an aspect ratio of 1 gives a better ellipse in the medium-resolution mode. This also causes the ellipse to be drawn faster. The start angle can be less than the end angle.

Examples :

```
10 SCREEN1: CIRCLE(100,100), 50
```

Draws a circle of radius 50, centered at graphics points 100x and 100y.

```
1 ' This will draw 17 circles
10 CLS
20 SCREEN 1
30 FOR R=160 TO 0 STEP-10
40 CIRCLE (160,100),R,,,,5/18
50 NEXT

10 'This will draw 5 circles
20 GOTO 160
50 IF VERT GOTO 100
60 CIRCLE (X,Y),R,C,,,0.7
70 FOR I = 1 TO 5
80 CIRCLE (X,Y),R,C,,,I*.2:NEXT I
90 IF VERT THEN RETURN
100 CIRCLE (X,Y),R,C,,,1.3
110 CIRCLE (X,Y),R,C,,,1.9
120 CIRCLE (X,Y),R,C,,,3.6
130 CIRCLE (X,Y),R,C,,,9.8
140 IF VERT GOTO 60
150 RETURN
160 CLS:SCREEN 1:COLOR 0,1:KEY OFF:VERT=0
170 X=160:Y=100:C=1:R=50:GOSUB 50
180 X=30:Y=30:C=2:R=30:GOSUB 50
190 X=30:Y=169:GOSUB 50
200 X=289:Y=30:GOSUB 50
210 X=289:Y=169:GOSUB 50
220 LINE (30,30)-(289,169),1
230 LINE (30,169)-(289,30),1
240 LINE (30,169)-(289,30),1,B
250 Z$=INKEY$: IF Z$="" THEN 250
```

CLEAR Command

Purpose:

To set all numeric variables to zero and all string variables to null, and to close all open files. Options set the end of memory and reserve the amount of string and stack space available for use by GW-BASIC.

Syntax:

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

Comments:

expression1 is a memory location that, if specified, sets the maximum number of bytes available for use by GW-BASIC.

expression2 sets aside stack space for GW-BASIC. The default is the previous stack space size. When you first execute GW-BASIC, the stack space is set to either 512 bytes or one-eighth of the available memory, whichever is smaller.

GW-BASIC allocates string space dynamically. An Out of String Space error occurs only if there is no free memory left for GW-BASIC to use.

The CLEAR command:

- Closes all files
- Clears all COMMON and user variables
- Resets the stack and string space
- Releases all disk buffers
- Turns off any sound
- Resets sound to music foreground
- Resets PEN to off
- Resets STRIG to off
- Disables ON ERROR trapping

Examples:

CLEAR

Zeros variables and nulls all strings.

CLEAR 32768

Zeros variables, nulls strings, protects memory above 32768, and does not change the stack space.

CLEAR „,2000

Zeros variables, nulls strings, allocates 2000 bytes for stack space, and uses all available memory in the segment.

CLEAR ,32768,2000

Zeros variables, nulls strings, protects memory above 32768, and allocates 2000 bytes for stack space.

CLOSE Statement

Purpose:

To terminate input/output to a disk file or a device.

Syntax:

`CLOSE [[#]file number [, [#]file number]...]`

Comments:

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

The association between a particular file or device and file number terminates upon execution of a CLOSE statement. You can reopen the file or device using the same or a different file number. If you use a different file number, you can reuse the terminated number for a new file or device.

A CLOSE statement with no file number specified closes all open files and devices.

A CLOSE statement sent to a file or device opened for sequential output writes the final buffer of output to that file or device.

The END, NEW, RESET, SYSTEM, or RUN and LOAD statements (without the *r* option) always close all files or devices automatically. STOP does not close files.

Examples:

```
250 CLOSE
```

Closes all open devices and files.

```
300 CLOSE 1,#2,#3
```

Closes all files and devices associated with file numbers 1,2, and 3.

CLS Statement

Purpose:

To clear the screen.

Syntax:

CLS

Comments:

If the graphics viewport is active, CLS clears only the viewport. If the graphics viewport is inactive, CLS clears the text window.

If the screen is in alpha mode, CLS clears the active page to the currently selected background color. See the SCREEN and COLOR statements.

If the screen is in graphics mode, CLS clears the entire screen buffer to the background color.

As well, you can press `CTRL HOME` to clear the screen or you can by changing the screen mode with the SCREEN function or the WIDTH statement.

CLS returns the cursor to the upper left corner of the screen, and sets the last point referenced to the center of the screen: 160,100 in medium resolution, or 320,100 in high resolution.

If the program previously used the VIEW statement, CLS clears only the last viewport specified.

Examples:

1 CLS

Clears the screen.

COLOR Statement

Purpose:

To select display colors.

Syntax:

```
COLOR [foreground ][,background ][,border]]  
COLOR [background ][,palette]]  
COLOR [foreground ][,background ]]
```

Comments:

COLOR lets you select screen foreground and background colors and, in Screen Mode 0 only, a border color. In Screen Mode 1, you can select two four-color palettes for use with graphics statements. The following table describes syntaxes and effects that apply to the various screen modes:

Mode	Effect
Screen Mode 0	<p>Modifies the current default text foreground and background colors, and the screen border. <i>foreground</i> must be an integer expression in the range 0-31. It determines foreground color in text mode, which is the default color of text. You can select sixteen colors (0-15). To select a blinking color, add 16 to the color number. For example, a blinking Color 7 is equal to 7 + 16, or 23. Thus, the legal integer range for foreground is 0-31.</p> <p><i>background</i> must be an integer expression in the range 0-7, and is the background color for each text character. Blinking colors are not permitted. The <i>border</i> color is an integer expression in the range 0-15 and is the color GW-BASIC uses when drawing the screen border. Blinking colors are not permitted.</p> <p>If you do not provide arguments for COLOR, the default for <i>background</i> and <i>border</i> is black (Color 0). The <i>foreground</i> default is as described in the SCREEN statement reference pages.</p>

Screen Mode 1

The **COLOR** statement has a unique syntax that includes a *palette* argument, which is an odd or even integer expression. This argument determines the set of display colors to use when displaying particular color numbers.

For hardware configurations that do not have an Enhanced Graphics Adapter (EGA), the default color settings for the *palette* parameter are equivalent to the following:

COLOR ,0 'Same as the next three PALETTE
 'statements
 '1 = green, 2 = red, 3 = yellow

COLOR ,1 'Same as the next three PALETTE
 'statements
 '1 = cyan, 2 = magenta,
 '3 = high intensity white

With the EGA, the default color settings for the palette parameter are equivalent to the following:

COLOR ,0 'Same as the next three PALETTE
 'statements

PALETTE 1,2 'Attribute 1 = Color 3 (green)

PALETTE 2,4 'Attribute 2 = Color 5 (red)

PALETTE 3,6 'Attribute 3 = Color 6 (brown)

COLOR ,1 'Same as the next three PALETTE
 'statements

PALETTE 1,3 'Attribute 1 = Color 3 (cyan)

PALETTE 2,5 'Attribute 2 = Color 5 (magenta)

PALETTE 3,7 'Attribute 3 = Color 15 (white)

Note that a **COLOR** statement overrides previous **PALETTE** statements.

Screen Mode 2

No effect. An **Illegal** function call message results if you use **COLOR** in this mode.

Screen Mode 7 through Screen Mode 10	In these modes, you cannot specify a <i>border</i> color. The graphics background is given by the <i>background</i> color number, which must be in the valid range of color numbers appropriate to the screen mode. See the SCREEN statement reference pages for more details. The <i>foreground</i> color argument is the default line drawing color. Arguments outside valid numeric ranges result in illegal function call errors.
--	---

You can set the foreground color to be the same as the background color, however, this makes the displayed characters invisible. The default background color is black (color number 0) for all display hardware configurations and all screen modes.

With the Enhanced graphics Adapter (EGA) installed, the PALETTE statement gives you flexibility in assigning different display colors to the actual color-number ranges for the *foreground*, *background*, and *border* colors discussed above. See the PALETTE statement reference pages for more details.

For more information, see CIRCLE, DRAW, LINE, PALETTE, PAINT, PRESET, PSET, and SCREEN.

Examples:

The following examples show the effects of various color statements in particular screen modes:

SCREEN 0	
COLOR 1, 2, 3	'foreground = 1, background = 2, border = 3
SCREEN 1	
COLOR 1,0	'foreground = 1, even palette number
COLOR 2,1	'foreground = 2, odd palette number
SCREEN 7	
COLOR 3,5	'foreground = 3, background = 5
SCREEN 8	
COLOR 6,7	'foreground = 6, background = 7
SCREEN 9	
COLOR 1,2	'foreground = 1, background = 2

COM(n) Statement

Purpose:

To enable or disable trapping of *communications* activity to the specified communications adapter.

Syntax:

```
COM(number) ON  
COM(number) OFF  
COM(number) STOP
```

Comments:

number is the number of the communications adapter 1 or 2.

Execute a COM(*number*) ON statement before an ON COM(*number*) statement to allow trapping. After COM(*number*) ON, if you specify a nonzero number in the ON COM(*number*) statement, GW-BASIC checks every new statement to see whether any characters have come in the communications adapter.

With COM(*number*) OFF, no trapping takes place, and all communications activity is lost.

With COM(*number*) STOP, no trapping takes place. However, GW-BASIC remembers any communication that takes place and it begins immediate trapping when it executes COM(*number*) ON.

COMMON Statement

Purpose:

To pass variables to a chained program.

Syntax:

COMMON *variables*

Comments:

variables are one or more variables, separated by commas, that you want to pass to the chained program.

Use the COMMON statement in conjunction with the CHAIN statement.

It is best to use COMMON statements at the beginning of a program but they can appear in any location.

You can use any number of COMMON statements in a program, but you cannot use the same variable in more than one COMMON statement. To pass all variables using the CHAIN statement, use the ALL option, and omit the COMMON statement.

To indicate that a variable is an array variable, place parentheses after the variable name.

Examples:

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

This example chains to program Prog3 on Disk Drive A:, and passes the Array D – along with the variables A, B, C, and String G\$ – to Prog3.

CONT Command

Purpose:

To continue program execution after a break.

Syntax:

CONT

Comments:

Resumes program execution after `CTRL BREAK`, `STOP`, or `END` halts a program. Execution continues at the point at which the break happened. If the break took place during an `INPUT` statement, execution continues after the prompt is redisplayed.

`CONT` is useful in debugging, in that it lets you set breakpoints with the `STOP` statement, modify variables using direct statements, continue program execution, or use `GOTO` to resume execution at a particular line number.

If you modify a program line, `CONT` is invalid.

COS Function

Purpose:

To return the cosine of the range of x .

Syntax:

$\text{COS}(x)$

Comments:

x must be in radians. COS is the trigonometric cosine function. To convert from degrees to radians, multiply by $\pi/180$.

BASIC calculates $\text{COS}(x)$ in single precision unless you use the /d switch when you execute GW-BASIC.

Examples:

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

10 PI=3.141593
20 PRINT COS(PI)
30 DEGREES=180
40 RADIANS=DEGREES*PI/180
50 PRINT COS(RADIANS)
RUN
-1
-1
OK
```

CSNG Function

Purpose:

To convert *number* to a single-precision number.

Syntax:

CSNG(*number*)

Comments:

number must be a numeric expression. (See the CINT and CDBL functions.)

Examples:

```
10 A# = 975.3421222#  
20 PRINT A#; CSNG(A#)  
  
RUN  
975.3421222 975.3421
```

CSRLIN Variable

Purpose:

To return the current line (row) position of the cursor.

Syntax:

row = CSRLIN

Comments:

row is a numeric variable receiving the value returned. The value returned is in the range 1-25.

The CSRLIN variable returns the vertical coordinate of the cursor on the active page. (See the SCREEN statement.)

column = POS(0) returns the column location of the cursor. The value returned is in the range 1-40 or 1-80, depending on the current screen width. See the POS function.

Examples:

```
10 ROW = CSRLIN
20 COLUMN = POS(0)
30 LOCATE 24,1
40 PRINT "HELLO"
50 LOCATE ROW,COLUMN

RUN
HELLO
```

The CSRLIN variable in Line 10 records the current row.

The POS function in Line 20 records the current column.

In Line 40, the PRINT statement displays the comment HELLO on the 24th line of the screen.

The LOCATE statement in Line 50 restores the position of the cursor to the original row and column.

CVI, CVS, CVD Functions

Purpose:

To convert string values to numeric values.

Syntax:

CVI(*2-byte string*)
CVS(*4-byte string*)
CVD(*8-byte string*)

Comments:

Your programs must convert strings from random-access disk files back into numbers if they are to be arithmetically manipulated.

CVI converts a 2-byte string to an integer. MKI\$ is its complement.

CVS converts a 4-byte string to a single-precision number. MKS\$ is its complement.

CVD converts an 8-byte string to a double-precision number. MKD\$ is its complement.

See MKI\$, MKS\$, and MKD\$.

Examples:

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

Line 80 reads a field from File #1 (the field read is defined in Line 70), and converts the first four bytes (N\$) into a single-precision number assigned to the variable Y.

Since a single-precision number can contain as many as seven ASCII characters (seven bytes), writing a file using MKS\$ conversion, and reading with the CVS conversion, lets you save as many as three bytes per number recorded on the storage medium. You can save even more bytes when double-precision numbers are required. MKD\$ and CVD conversions could be used in this case.

DATA Statement

Purpose:

To store the numeric and string constants accessed by the program READ statement(s).

Syntax:

DATA *constants*

Comments:

constants are numeric constants in any format (fixed point, floating-point, or integer), separated by commas. You cannot include expressions in the list.

Surround string constants in DATA statements with double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, you do not need quotation marks.

DATA statements are not executable. You can place them anywhere in the program. A DATA statement can contain as many constants as fit on a line (separated by commas), and any number of DATA statements can be used in a program.

READ statements access the DATA statements in order (by line number). The data contained therein can 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. The variable type (numeric or string) in the READ statement must agree with the corresponding constant in the DATA statement, or a Type Mismatch error occurs.

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

For further information and examples, see the RESTORE statement and the READ statement.

Examples:

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

The previous program segment reads the values from the DATA statements into Array A. After execution, the value of A(1) is 3.08, and so on. You can place the DATA statement (lines 110-120) anywhere in the program. You can even place them ahead of the READ statement.

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

```
RUN  
CITY      STATE      ZIP  
DENVER,   COLORADO   80211
```

The previous program reads string and numeric data from the DATA statement in Line 30.

DATE\$ Statement and Variable

Purpose:

To set or retrieve the current date.

Syntax:

As a statement:

`DATE$ = v$`

As a variable:

`v$ = DATE$`

Comments:

`v$` is a valid string literal or variable.

`v$` can be any of the following formats when assigning the date:

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

If `v$` is not a valid string, a **Type Mismatch** error results. Previous values are retained.

If any values are out of range or missing, GW-BASIC issues an **Illegal Function Call** error. GW-BASIC retains any previous date.

BASIC gets the current date (as assigned when the operating system was initialized) and assigns it to the string variable if `DATE$` is the expression in a **LET** or **PRINT** statement.

The current date is stored if `DATE$` is the target of a string assignment.

With `v$ = DATE$`, `DATE$` returns a 10-character string in the form *mm-dd-yyyy*. *mm* is the month (01-12), *dd* is the day (01-31), and *yyyy* is the year (1980-2099).

Examples:

```
V$ = DATE$  
OK  
PRINT V$  
01-01-1985
```

DEF FN Statement

Purpose:

To define and name a function you create.

Syntax:

DEF FN*name*[*arguments list*]*expression*

Comments:

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

arguments list consists of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

expression is an expression that performs the operation of the function. It is limited to one statement.

In the DEF FN statement, arguments serve only to define the function; they do not affect program variables that have the same name. A variable name you use in a function definition might or might not appear in the argument. If it does, GW-BASIC supplies the value of the parameter when it calls the function. Otherwise, GW-BASIC uses the current value of the variable.

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

User-defined functions can be numeric or string. If you specify a type in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If you specify a type in the function name and the argument type does not match, a Type Mismatch error occurs.

You can define a user-defined function more than once in a program by repeating the DEF FN statement.

You must execute a DEF FN statement before you can call the function it defines. If you call a function before it has been defined, an Undefined User Function error occurs.

DEF FN is illegal in the direct mode.

Recursive functions are not supported in the DEF FN statement.

Examples:

```
.  
.   
.   
400 R=1:S=2  
410 DEF FNAB(X,Y)=X^3/Y^2  
420 T=FNAB(R,S)  
.   
.   
.
```

In the previous example, Line 410 defines the user-defined function FNAB. The function is called in Line 420. When executed, the variable T contains the value R^3 divided by S^2 , or .25.

DEFINT/SNG/DBL/STR Statements

Purpose:

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

Syntax:

DEF*type* *letters*

Comments:

type is INT (integer), SNG (single-precision number), DBL (double-precision number), or STR (string of 0-255 characters).

letters are letters (separated by commas) or a range of letters of the alphabet.

A DEF*type* statement declares that variable names beginning with the letter(s) specifies that type of variable. However, a type declaration character (% , ! , # , \$) always takes precedence over a DEF*type* statement.

If no type declaration statements are encountered, GW-BASIC assumes all variables are single-precision.

Examples:

```
10 DEFDBL L-P
```

In the previous example, all variables beginning with the letter L, M, N, O, or P become double-precision variables.

```
10 DEFSTR A
20 A = "120#"
```

In the previous example, all variables beginning with the letter A become string variables. The \$ declaration is unnecessary in this example.

```
10 DEFINT I-N,W-Z
20 W$="120#"
```

In the previous example, all variables beginning with the letter I, J, K, L, M, N, W, X, Y, or Z become integer variables. W\$ in Line 20 establishes a string variable beginning with the letter W. However, the variable W remains an integer elsewhere in the program.

DEF SEG Statement

Purpose:

To assign the current segment address for reference by a subsequent BLOAD, BSAVE, CALL, PEEK, POKE, or USR.

Syntax:

DEF SEG [*address*]

Comments:

address is a numeric expression in the range 0-65535.

BASIC saves the specified address for use as the segment required by the BLOAD, BSAVE, PEEK, POKE, and CALL statements.

If you enter any value outside the address range (0-65535) GW-BASIC returns an Illegal Function Call error. GW-BASIC retains the previous value.

If you omit the *address* option, GW-BASIC sets the segment to be used to its data segment (DS). This is the initial default value.

For a BLOAD, BSAVE, PEEK, POKE, or CALL statement, the value is shifted left four bits to form the code segment address for the subsequent call instruction. (This is done by the microprocessor, not by GW-BASIC.) See the BLOAD, BSAVE, CALL, PEEK, and POKE statements.

BASIC does not perform additional checking to assure that the resultant segment address is valid.

Examples:

```
10 DEF SEG = &HB800
```

Sets segment to screen buffer.

```
20 DEF SEG
```

Restores segment to GW-BASIC DS.

Note: DEF and SEG must be separated by a space. Otherwise, GW-BASIC interprets the statement DEFSEG = 100 to mean "assign the value 100 to the variable DEFSEG".

DEF USR Statement

Purpose:

To specify the starting address of an assembly language subroutine to be called from memory by the USR function.

Syntax:

DEF USR[*number*] = *integer*

Comments:

number can be any digit from 0 to 9. The digit corresponds to the USR routine address being specified. If you omit *number*, GW-BASIC assumes DEF USR0.

integer is the offset address of the USR routine. If more than 10 USR routines are required, DEF USR[*number*] can appear in the program as many times as necessary to redefine the USR[*number*] starting address.

Add the current segment value to the integer to get the starting address of user routine.

When GW-BASIC calls an assembly language subroutine, it pauses and transfers control to the assembly language program. When the execution of the assembly language program is complete, the system returns control the GW-BASIC program at the point of interruption.

Examples:

```
.  
.   
.   
190 DEF SEG = 0  
200 DEF USR0 = 24000  
210 X = USR0(Y ^ 2/2.82)  
.   
.   
. 
```

Lines 190 and 200 set the absolute address.

Line 210 calls the USR routine located at that address, and passes the integer value of the expression contained within the parentheses to the user program (see USR).

Note: This statement provides compatibility with other GW-BASIC implementations. However, you should use the more versatile CALL statement if this downward compatibility is not necessary.

DELETE Command

Purpose:

To delete program lines or groups of lines.

Syntax:

```
DELETE [line1]-[line2]  
DELETE line1
```

Comments:

line1 is the first line to be deleted.

line2 is the last line to be deleted.

GW-BASIC always returns to command level after DELETE completes execution. Unless you give at least one line number, an Illegal Function Call error occurs.

You can use the period (.) to substitute for either line number to indicate the current line.

Examples:

```
DELETE 40
```

Deletes Line 40.

```
DELETE 40-100
```

Deletes Lines 40 through 100, inclusively.

```
DELETE -40
```

Deletes all lines up to and including Line 40.

```
DELETE 40-
```

Deletes all lines from Line 40 to the end of the program.

DIM Statement

Purpose:

To specify the maximum values for array variable subscripts and allocate storage accordingly.

Syntax:

DIM variable(subscripts)[,variable(subscripts)]...

Comments:

If you use an array variable name without a DIM statement, GW-BASIC assumes the maximum value of its subscript(s) is 10. If you use a subscript greater than the maximum specified, a Subscript out of range error occurs.

The maximum number of dimensions for an array is 255.

The minimum value for a subscript is always 0 unless otherwise specified with the OPTION BASE statement.

Once you dimension an array, you cannot redimension it within the program without first executing a CLEAR or ERASE statement.

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

Examples:

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

The program lines read 21 DATA statements elsewhere in the program and assigns their values to A(0) through A(20), sequentially and inclusively. If the A array is single precision (default accuracy) then Line 10 allocates 84 bytes of memory to this array (4 bytes times 21 elements).

DRAW Statement

Purpose:

To draw a figure.

Syntax:

DRAW *string expression*

Comments:

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

The DRAW statement is valid only in graphics mode.

Movement Commands: Each of the following movement commands begins movement from the current graphics position. This is usually the coordinate of the last graphics point plotted with another GML command, LINE, or PSET. The current position defaults to the center of the screen (160,100 in medium resolution; 320,100 in high resolution) when you run a program. Movement commands move for a distance of scale factor $*n$, where the default for n is 1; thus they move one point if you omit n and GW-BASIC uses the default scale factor.

Command	Moves
<i>Un</i>	up
<i>Dn</i>	down
<i>Ln</i>	left
<i>Rn</i>	right
<i>En</i>	diagonally up and right
<i>Fn</i>	diagonally down and right
<i>Gn</i>	diagonally down and left
<i>Fn</i>	diagonally up and left

The M command moves as specified by the following argument:

Mx,y Move absolute or relative. If you precede *x* with a + or -, GW-BASIC adds *x* and *y* to the current graphics position, and connected to the current position by a line. Otherwise, it draws a line to point *x,y* from the current position.

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

B Move, but plot no points.

N Move, but return to original position when done.

The following commands are also available:

An Set angle *n*. *n* can range from 0 to 3, where 0 is 0°, 1 is 90°, 2 is 180°, and 3 is 270°. Figures rotated 90° or 270° are scaled so that they appear the same size as 0° or 180° on a monitor screen with the standard aspect ratio of 4:3.

TAn Turn angle *n* degrees. *n* can be any value from negative 360° to positive 360°. If the value specified by *n* is positive, it turns the angle counterclockwise. If the value specified by *n* is negative, it turns the angles clockwise.

Cn Set color *n*. See the **COLOR**, **PALETTE**, and **SCREEN** statements for discussions of valid colors, numbers, and attributes.

Sn Set scale factor. *n* can range from 1 to 255. *n* is divided by 4 to derive the scale factor. The scale factor is multiplied by the distances given with the **U**, **D**, **L**, **R**, **E**, **F**, **G**, **H**, or relative **M** commands to get the actual distance traveled. The default for **S** is 4.

- Xstring, variable* Execute substring. This command executes a second substring from a string, much like GOSUB. One string executes another, which executes a third, and so on.
- string* is a variable assigned to a string of movement commands.
- Ppaint, boundary* Specifies the colors for a graphics figure and creates a filled-in figure.
- paint* specifies what color you want the figure filled in with.
- boundary* specifies the border color (outline).
- See the COLOR, PALETTE, and SCREEN statements for discussions of valid colors, numbers, and attributes.
- You must specify values for both *paint* and *boundary* when using P.
- This command (*Ppaint, boundary*) does not paint color tiling.
- numeric arguments Numeric arguments can be constants such as "123" or "*=variable*";, where *variable* is the name of a variable.
- When you use the second syntax, "*=variable*";, you must use the semicolon. Otherwise, the semicolon is optional between commands.
- You can also specify variables using VARPTR\$(*variable*).

Examples:

To draw a box in medium resolution:

```
10 SCREEN 1
20 A = 20
30 DRAW "U = A; R = A; D = A; L = A;"
```

The aspect ratio to draw a square on a standard screen is 4:3. To draw a 96-pixel-wide square on a 640 x 200 pixel screen (Screen Mode 2), do the following calculations:

Horizontal value = 96

Vertical value = $96 * (200/640) * (4/3)$

or

Vertical value = 40

Horizontal value = $40 * (640/200) * (3/4)$

Horizontal screen equals 4/3 of the vertical screen.

To draw a triangle in medium resolution:

```
10 CLS
20 SCREEN 1
30 PSET (60,125)
40 DRAW "E100; F100; L199"
```

Compiler Note: The "=" variable;" is not supported by the BASIC compiler. The only way to specify a variable when using the GW-BASIC Compiler is to use `VARPTR$(variable)`.

EDIT Command

Purpose:

To display a specified line and to position the cursor under the first digit of the line number so that you can edit the line.

Syntax:

```
EDIT line  
EDIT .
```

Comments:

line is the number of a line existing in the program.

A period (.) refers to the current line. The following command enters EDIT at the current line:

```
EDIT .
```

When you enter a line, it becomes the current line.

The current line is always the last line referred to by an EDIT statement, LIST command, or error message.

If *line* refers to a line that does not exist in the program, an Undefined Line Number error occurs.

Examples:

```
EDIT 150
```

Displays Line 150 for editing.

END Statement

Purpose:

To terminate program execution, close all files, and return to the command level.

Syntax:

END

Comments:

You can place END statements anywhere in a program.

Unlike the STOP statement, END does not cause a **Break in line xxxx** message to be printed.

Because GW-BASIC always returns to command level after it executes a program, you do not have to include an END statement at the end of a program .

END closes all files.

Examples:

```
520 IF K1 > 1000 THEN END ELSE GOTO 20
```

Ends the program and returns to command level whenever the value of K exceeds 1000.

ENVIRON Statement

Purpose:

To allow you to modify parameters in GW-BASIC's environment string table. This might be to change the path parameter for a child process, or to pass parameters to a child by inventing a new environment parameter. (See ENVIRON\$, SHELL, and the MS-DOS utilities PATH command.)

Syntax:

ENVIRON *string*

Comments:

string is a valid string expression containing the new environment string parameter.

string must be of the following form

*parm*id = *text*

where *parm*id is the name of the parameter, such as PATH.

ENVIRON takes everything to the left of the first blank or equal sign as the *parm*id; it takes everything following the blank or equal sign as *text*. Therefore, you must separate *parm*id from *text* by an equal sign or by a blank.

text is the new parameter text. If *text* is a null string or consists only of a single semicolon, then ENVIRON removes the parameter (including *parm*id =) from the environment string table and compresses the table. *text* must not contain any embedded blanks.

If *parm*id does not exist, then ENVIRON adds *string* at the end of the environment string table.

If *parm*id does exist, ENVIRON deletes it, compresses the environment string table, and adds the new string at the end.

Examples:

Assuming the environment string table is empty, the following statement creates a default path to the Root directory on Disk A:

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

If your work subdirectory is John, you can get DEBUG from the Root.

You can add a new parameter:

```
ENVIRON "COMSPEC = A:\COMMAND.COM"
```

The environment string table now contains

```
PATH = A:\;COMSPEC = A:\COMMAND.COM
```

You can change the path to a new value:

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

You can append the path parameter by using the ENVIRON\$ function with the ENVIRON statement:

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

Finally, delete the parameter COMSPEC:

```
ENVIRON "COMSPEC = ;"
```

The environment string table now contains

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

ENVIRON\$ Function

Purpose:

To allow you to retrieve the specified environment string from the environment table.

Syntax:

```
v$ = ENVIRON$(parm)  
v$ = ENVIRON$(nthparm)
```

Comments:

parm is a valid string expression containing the parameter to search for.

nthparm is an integer expression in the range 1-255.

If you use a string argument, ENVIRON\$ returns a string containing the text following *parm* = from the environment string table.

If ENVIRON\$ does not find *parm*, then it returns a null string.

If you use a numeric argument, ENVIRON\$ returns a string containing the *nth* parameter from the environment string table.

If there is no *nth* parameter, then ENVIRON\$ returns a null string.

The ENVIRON\$ function distinguishes between upper- and lower-case.

Examples:

The following lines:

```
10 ENVIRON "PATH=A:\SALES;A:\ACCOUNTING;B:\MKT:"  
   'Create entry  
20 PRINT ENVIRON$("PATH") 'Print entry
```

will print the following string:

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

The following line prints the first string in the environment:

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

The following program saves the environment string table in an array so that it can be modified for a child process. After the child process finishes processing, the program restores the environment.

```
10 DIM ENVTBL$(10) "  
20 NPARMS = 1  
30 WHILE LEN(ENVIRON$(NPARMS)) 0  
40 ENVTBL$ (NPARMS) = ENVIRON$(NPARMS)  
50 NPARMS = NPARMS + 1  
60 WEND  
70 NPARMS = NPARMS-1  
72 WHILE LEN(ENVIRON$(1))0  
73 A$ = MID$(ENVIRON$(1),1,INSTR (ENVIRON$(1),"="))  
74 ENVIRON A$ + ";"  
75 WEND  
90 ENVIRON "MYCHILDPARM1= SORT BY NAME"  
100 ENVIRON "MYCHILDPARM2= LIST BY NAME"  
.  
.  
.  
1000 SHELL "MYCHILD"RUNS "MYCHILD.EXE"  
1002 WHILE LEN(ENVIRON$(1))0  
1003 A$ = MID$(ENVIRON$(1),1,INSTR(ENVIRON$ (1),"="))  
1004 ENVIRON A$ + ";"  
1005 WEND  
1010 FOR I= 1 TO NPARMS  
1020 ENVIRON ENVTBL$(I)  
1030 NEXT I  
.  
.  
.
```

The DIM statement in Line 10 assumes no more than 10 parameters are to be accessed.

In Line 20, the initial number of parameters is established as 1.

In Lines 30 through 70, a series of statements adjusts and corrects the parameter numbers.

Line 71 deletes the present environment.

Lines 72 through 80 create a new environment. Line 74 deletes the string.

Lines 80 through 100 store the new environment.

Lines 1000 through 1030 repeat the procedure by deleting the present environment and restoring the parameters established in the first part of the program.

EOF Function

Purpose:

To return -1 (true) when the end of a sequential or a communications file has been reached, or to return 0 if the end of file (EOF) has not been found.

Syntax:

$v = \text{EOF}(\text{file number})$

Comments:

If a program attempts a GET past the end of the file, EOF returns -1. You can use this feature to find the size of a file using a binary search or other algorithm. With communications files, a -1 indicates that the buffer is empty.

Use EOF to test for end of file while inputting to avoid Input Past End errors.

Examples:

```
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  
100 END
```

These lines read the contents of the file named Data into the M array until the end of the file is reached; then, the program branches to Line 100.

ERASE Statement

Purpose:

To eliminate arrays from a program.

Syntax:

ERASE *list of array variables*

Comments:

You can redimension arrays after they are erased, or use the memory space previously allocated to the array for other purposes.

If you attempt to redimension an array without first erasing it, an error occurs.

Examples:

```
200 DIM B (250)
.
.
.
450 ERASE A,B
460 DIM B(3,4)
```

Arrays A and B are eliminated from the program. The B array is redimensioned to a 3-column by 4-row array (12 elements), all elements of which are set to zero values.

ERDEV and ERDEV\$ Variables

Purpose:

To return the actual value (ERDEV) of a device error and the name of the device (ERDEV\$) causing the error.

Syntax:

ERDEV
ERDEV\$

Comments:

ERDEV contains the error code from interrupt 24H in the lower 8 bits. Bits 8 to 15 from the attribute word in the device header block are mapped directly into the upper 8 bits.

ERDEV\$ contains the 8-byte character device name if the error was on a character device. It contains the 2-byte block device name (A:, B:, etc.) if the device is not a character device.

Examples:

Installed device driver Lpt2: caused a Printer out of paper error via interrupt 24H.

ERDEV contains the error Number 9 in the lower 8 bits, while the upper 8 bits contain the upper byte of the device header word attributes.

ERDEV\$ contains "LPT2: ".

ERR and ERL Variables

Purpose:

To return the error code (ERR) and line number (ERL) associated with an error.

Syntax:

```
v = ERR
v = ERL
```

Comments:

The variable ERR contains the error code for the last occurrence of an error. All the error codes and their definitions are listed in Appendix A of the *Tandy GW-BASIC User's Guide*.

The variable ERL contains the line number of the line in which the error was detected.

You would usually use the ERR and ERL variables in IF-THEN, ON ERROR...GOTO, or GOSUB statements to direct program flow in error trapping.

If the statement that caused the error is a direct mode statement, ERL contains 65535. To see whether an error occurred in a direct mode statement, use a line of the following form:

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

Otherwise, use

```
10 IF ERR = error code THEN...GOSUB 4000
20 IF ERL = line number THEN...GOSUB 4010
```

Note: If the line number is not on the right side of the relational operator, RENUM cannot renumber it.

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

ERROR Statement

Purpose:

To simulate the occurrence of an error or to allow you to define error codes.

Syntax:

ERROR *integer expression*

Comments:

The value of *integer expression* must be greater than 0 and less than 255.

If the value of *integer expression* equals an error code already in use by GW-BASIC, the ERROR statement simulates the occurrence of that error, and the corresponding error message is printed.

A user-defined error code must use a value greater than any used by the GW-BASIC error codes. There are 76 GW-BASIC error codes at present. It is preferable to use a code number high enough to remain valid when more error codes are added to GW-BASIC.

User-defined error codes can be used in an error-trapping routine.

If an ERROR statement specifies a code for which no error message has been defined, GW-BASIC responds with Unprintable Error.

If you execute an ERROR statement for which there is no error-trapping routine, an error message prints and execution halts.

For a complete list of the error codes and messages already defined in GW-BASIC, refer to Appendix A in the *Tandy GW-BASIC User's Guide*.

Examples:

The following examples simulate Error 15 (the code for String too long):

```
LIST
10 S=10
20 T=5
30 ERROR S+T
40 END

RUN
String too long in 30
```

Or, in direct mode:

```
Ok
ERROR 15          (you type this line)
String too long   (GW-BASIC types this line)
The following example includes a user-defined error code message.
```

```
.
.
.
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
.
.
.
```

EXP Function

Purpose:

To return e (the base of natural logarithms) to the power of *number*.

Syntax:

EXP(*number*)

Comments:

number must be less than 88.02969.

If EXP overflows, an **Overflow** error appears; machine infinity with the appropriate sign is supplied as the result, and execution continues.

EXP(*number*) is calculated in single precision, unless you use the /d switch when executing GW-BASIC.

Examples:

```
10 NUMBER = 5
20 PRINT EXP(NUMBER-1)

RUN
54.59815
```

Prints the value of e to the 4th power.

EXTERR Function

Purpose:

To return extended error information.

Syntax:

EXTERR(*number*)

Comments:

EXTERR returns “extended” error information provided by versions of DOS 3.0 and greater. For versions of DOS earlier than 3.0, EXTERR always returns zero. The single integer argument must be in the range 0-3 as follows:

Value of <i>number</i>	Return Value
0	Extended error code
1	Extended error class
2	Extended error suggested action
3	Extended error location

The values returned are not defined by GW-BASIC but by DOS. Refer to the *MS-DOS Programmer's Reference* (Version 3.0 or later) for a description of the values returned by the DOS extended error function.

BASIC retrieves and saves the extended error code each time appropriate DOS functions are performed. Thus, when an EXTERR function call is made, these saved values are returned.

FIELD Statement

Purpose:

To allocate space for variables in a random file buffer.

Syntax:

FIELD [#] *file number*, *width* AS *stringvar* [, *width* AS *stringvar*]...

Comments:

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

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

stringvar is a string variable to be used for random file access.

You must execute a FIELD statement before you can:

- get data out of a random buffer after a GET statement
- enter data before a PUT statement

For example, the following line 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 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

FIELD only allocates space; it does not place any data in the random file buffer.

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

You can execute any number of FIELD statements for the same file, and all FIELD statements executed are 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 fielded, 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 string space. See the LSET/RSET and GET statements.

FILES Command

Purpose:

To print the names of the files residing on the specified drive.

Syntax:

FILES [*pathname*]

Comments:

If you omit *pathname*, the command lists all files in the current directory of the selected drive. *pathname* can contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension matches any file or any extension.

This syntax also displays the name of the directory and the number of bytes in the file. When a tree-structured directory is used, two special symbols also appear.

Subdirectories are denoted by DIR following the directory name.

Examples:

```
FILES
FILES "*.BAS"
FILES "B:*.*)"
FILES "TEST?.BAS"
```

FILES now allows pathnames. The directory for the specified path is displayed. If an explicit path is not given, the current directory is assumed.

```
FILES "ACCTS\"
```

Lists all files in the directory named Accts. Include the backslash when specifying the directory named.

FILES "B:ACCTS*.PAY"

Lists all files in the directory named Accts that are on the diskette in Drive B and have the extension of .pay.

FIX Function

Purpose:

To truncate *number* to a whole *number*.

Syntax:

FIX(*number*)

Comments:

FIX does not round off numbers, it simply eliminates the decimal point and all characters to the right of the decimal point.

FIX(*number*) is equivalent to $\text{SGN}(x) * \text{INT}(\text{ABS}(x))$. The major difference between FIX and INT is that FIX does not return the next lower number for negative *number*.

FIX is useful in modulus arithmetic.

Examples:

```
PRINT FIX(58.75)
      58
```

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

FOR and NEXT Statements

Purpose:

To execute a series of instructions a specified number of times in a loop.

Syntax:

```
FOR variable = x TO y [STEP z] .  
.  
.  
NEXT [variable][,variable...]
```

Comments:

variable is used as a counter.

x, *y*, and *z* are numeric expressions.

STEP *z* specifies counter increment for each loop.

The first numeric expression, or *x*, is the initial value of the counter. The second numeric expression, or *y*, is the final value of the counter.

BASIC executes program lines following the FOR statement until it encounters the NEXT statement. Then, it increments the counter by the amount specified by STEP.

If you do not specify STEP, GW-BASIC assumes the increment is 1.

BASIC performs a check to see whether the value of the counter is now greater than *y*. If it is not greater, GW-BASIC branches back to the statement after the FOR statement, and the process is repeated. If it is greater, GW-BASIC continues execution at the statement following the NEXT statement. This is a FOR-NEXT loop.

BASIC skips the body of the loop if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

If STEP is negative, GW-BASIC sets the final value of the counter to be less than the initial value. It decreases the counter each time through the loop, and executes the loop until the counter is less than the final value.

Nested Loops: You can nest FOR-NEXT loops; that is, you can place a FOR-NEXT loop within the context of 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 the NEXT statement for the outside loop.

If nested loops have the same end point, you can use a single NEXT statement for all of them.

You can omit the *variable(s)* in the NEXT statement, in which case the NEXT statement matches the most recent FOR statement.

If GW-BASIC encounters a NEXT statement before its corresponding FOR statement, a NEXT without FOR error is issued and execution is terminated.

Examples:

The following example prints integer values of the variable I% from 1 to 10 in steps of 2. For fastest execution, I is declared as an integer by the % sign.

```
10 K=10
20 FOR I%=1 TO K STEP 2
30 PRINT I%
.
.
.
60 NEXT
RUN
1
3
5
7
9
```

In the following example, the loop does not execute because the initial value of the loop exceeds the final value. Nothing is printed by this example.

```
10 R=0
20 FOR S=1 TO R
30 PRINT S
40 NEXT S
```

In the next example, the loop executes 10 times. The final value for the loop variable is always set before the initial value is set.

```
10 S=5
20 FOR S=1 TO S+5
30 PRINT S;
40 NEXT
```

RUN

1 2 3 4 5 6 7 8 9 10

FRE Function

Purpose:

To return the number of available bytes in allocated string memory.

Syntax:

`FRE(x$)`
`FRE(x)`

Comments:

Arguments (x\$) and (x) are dummy arguments.

Before `FRE(x$)` returns the amount of space available in allocated string memory, GW-BASIC initiates a “garbage collection” activity. Data in string memory space is collected and reorganized, and unused portions of fragmented strings are discarded to make room for new input.

If `FRE` is not used, GW-BASIC initiates an automatic garbage collection activity when all string memory space is used up. GW-BASIC does not initiate garbage collection until all free memory has been used. Garbage collection might take 1 to 1 1/2 minutes.

`FRE(“”)`, or any string, forces a garbage collection before returning the number of free bytes. Therefore, using `FRE(“”)` periodically results in shorter delays for each garbage collection.

You cannot use `CTRL` `BREAK` during this housecleaning process.

Examples:

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

Your computer might return a different value.

GET Statement (Files)

Purpose:

To read a record from a random disk file into a random buffer.

Syntax:

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

Comments:

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

record is the number of the record, in the range 1-16,777,215.

If you omit *record*, GET reads into the buffer the next record (after the last GET).

After a GET statement, use INPUT# and LINE INPUT# to read characters from the random file buffer.

You can also use GET for communications files. *record* is the number of bytes to be read from the communications buffer. *record* cannot exceed the buffer length set in the OPEN COM(*n*) statement.

Examples:

```
10 OPEN "R",1,"A:VENDOR.FIL"  
20 FIELD 1,30 AS VENDNAMES$,20 AS ADDR$,15 AS  
   CITY$  
30 GET 1  
40 PRINT VENDNAMES$,ADDR$,CITY$  
50 CLOSE 1
```

This example opens the file Vendor.fil for random access, with fields defined in Line 20. In Line 30, the GET statement reads a record into the file buffer. Line 40 displays the information from the record just read. Line 50 closes the file.

GET Statement (Graphics)

Purpose:

To transfer graphics images from the screen.

Syntax:

GET (x1,y1)-(x2,y2), *array name*

Comments:

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

The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array. The rectangle is defined the same way as the rectangle drawn by the LINE statement using the ,B option.

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

The storage format in the array is as follows:

- 2 bytes given *x* dimension in bits
- 2 bytes given *y* dimension in bits
- the array data itself

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

$$4 + \text{INT}((x * \text{bits per pixel} + 7) / 8) * y$$

See the SCREEN statement for bits-per-pixel values for different screen modes.

The number of bytes per element of an array are as follows:

- 2 for integer
- 4 for single-precision
- 8 for double-precision

The number of bytes required to get a 10 by 12 image into an integer array is $4 + \text{INT}((10*2 + 7)/8)*12$, or 40 bytes. An integer array with at least 20 elements is necessary.

If `OPTION BASE` equals zero, you can use an integer array to examine the *x* and *y* dimensions and the data. The *x* dimension is in element 0 of the array, and the *y* dimension is in element 1. Integers are stored low byte first, then high byte, but data is transferred high byte first (leftmost), then low byte.

It is possible to get an image in one mode and put it in another, although the effect might be quite strange because of the way points are represented in each mode.

Examples:

```
10 CLS:SCREEN 1
20 PSET(130,120)
30 DRAW "U25;E7;R20;D32;L6;U12;L14"
40 DRAW "D12;L6":PSET(137,102)
50 DRAW "U4;E4;R8;D8;L12"
60 PSET(137,88)
70 DRAW "E4;R20;D32;G4":PAINT(139,87)
80 DIM A(500)
90 GET (125,130)-(170,80),A
100 FOR I=1 TO 1000:NEXT I
110 PUT (20,20),A,PSET
120 FOR I=1 TO 1000:NEXT I
130 GET (125,130)-(170,80),A
140 FOR I=1 TO 1000:NEXT I
150 PUT (220,130),A,PRESET
```

GOSUB...RETURN Statement

Purpose:

To branch to, and return from, a subroutine.

Syntax:

```
GOSUB line  
.  
.  
.  
RETURN [line]
```

Comments:

line is the first line number of the subroutine.

You can call a subroutine any number of times in a program, and you can call a subroutine from within another subroutine. Such nesting of subroutines is limited only by available memory.

A RETURN statement in a subroutine causes GW-BASIC to return to the statement following the most recent GOSUB statement. A subroutine can contain more than one RETURN statement, should logic dictate a RETURN at different points in the subroutine.

Subroutines can appear anywhere in the program but must be readily distinguishable from the main program.

To prevent inadvertent entry, precede the subroutine by a STOP, END, or GOTO statement to direct program control around the subroutine.

Examples:

```
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
```

The END statement in Line 30 prevents re-execution of the subroutine.

GOTO Statement

Purpose:

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

Syntax:

GOTO line

Comments:

line is any valid line number within the program.

If *line* is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after *line*.

Examples:

```
10 READ R
20 PRINT "R =";R;
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12

RUN
R = 5      AREA = 78.5
R = 7      AREA = 153.86
R = 12     AREA = 452.16
Out of data in 10
```

The Out of data advisory is generated when the program attempts to read a fourth DATA statement (which does not exist) in Line 60.

HEX\$ Function

Purpose:

To return a string that represents the hexadecimal value of the numeric argument.

Syntax:

`v$ = HEX$(number)`

Comments:

HEX\$ converts decimal values in the range -32768 to +65535 into a hexadecimal string expression in the range 0 to FFFF.

Hexadecimal numbers are numbers to the base 16, rather than base 10 (decimal numbers). Appendices B and F in the *Tandy GW-BASIC User's Guide* contain more information on hexadecimals and their equivalents.

BASIC rounds *number* to an integer before evaluating HEX\$(*number*). See the OCT\$ function for octal conversions.

If *number* is negative, HEX\$ uses two's (binary) complement form.

Examples:

```
10 CLS:INPUT "INPUT DECIMAL NUMBER";X
20 A$=HEX$(X)
30 PRINT X "DECIMAL IS "A$" HEXADECIMAL"

RUN
INPUT DECIMAL NUMBER? 32
32 DECIMAL IS 20 HEXADECIMAL
```

IF Statement

Purpose:

To make a decision regarding program flow based on the result returned by an expression.

Syntax:

```
IF expression[,] THEN statement(s)[,][ELSE statement(s)]  
IF expression[,] GOTO line number [,] ELSE statement(s)
```

Comments:

If the result of *expression* is nonzero (logical true), GW-BASIC executes the THEN statement or the GOTO line number.

If the result of *expression* is zero (false), GW-BASIC ignores the THEN statement or the GOTO line number. The ELSE line number, if present, is executed. Otherwise, execution continues with the next executable statement. A comma is allowed before THEN and ELSE.

You can follow THEN and ELSE by either a line number for branching or one or more statements to be executed.

Always follow GOTO with a line number.

If the statement does not contain the same number of ELSE's and THEN's, GW-BASIC matches each ELSE with the closest unmatched THEN. For example:

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

Does not print A < > C when A < > B.

If you follow an IF-THEN statement by a line number in the direct mode, GW-BASIC returns an Undefined line number error, unless a program had previously entered a statement with the specified line number.

Because IF-THEN-ELSE is all one statement, the ELSE clause cannot be on a separate line. All three clauses must be on one line.

To Test Equality: When using IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value might not be exact. Therefore, test against the range over which the accuracy of the value might vary.

For example, to test a computed variable A against the value 1.0, use the following statement:

```
IF ABS (A-1.0) < .0E-6 THEN ...
```

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

Examples:

The following statement gets record number N, if N is not zero.

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

In the following example, a test determines whether N is greater than 10 and less than 20. If N is within this range, DB is calculated and execution branches to Line 300. If N is not within this range, execution continues with Line 110.

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

```
.  
. .  
.
```

The next statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the terminal.

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

INKEY\$ Variable

Purpose:

To return one character read from the keyboard.

Syntax:

`v$ = INKEY$`

Comments:

If no character is pending in the keyboard buffer, INKEY\$ returns a null string (length zero).

If several characters are pending, GW-BASIC returns only the first. The string is one or two-characters in length.

BASIC uses two-character strings to return the extended codes described in Appendix B of the *Tandy GW-BASIC User's Guide*. The first character of a two-character code is zero.

BASIC does not display any characters on the screen, and passes all characters to the program except codes from the following keys:

`CTRL` `BREAK`

`CTRL`

`NUM LOCK`

`CTRL` `ALT`

`DELETE`

`CTRL` `PRINT SCRN`

`PRINT SCRN`

Examples:

```
10 CLS: PRINT "PRESS RETURN "
20 TIMELIMIT% = 1000
30 GOSUB 1010
40 IF TIMEOUT% THEN PRINT "TOO LONG" ELSE PRINT
   "GOOD SHOW"
50 PRINT RESPONSE$
60 END
.
.
.
1000 REM TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR N%=1 TO TIMELIMIT%
1030 A$=INKEY$:IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0:RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT N%
1070 TIMEOUT%=1:RETURN
```

When this program is executed, and if you press **ENTER** before 1000 loops are completed, then "GOOD SHOW" is printed on the screen. Otherwise, "TOO LONG" is printed.

Since an INKEY\$ statement scans the keyboard only once, place INKEY\$ statements within loops to provide adequate response times for the operator.

INP Function

Purpose:

To return the byte read from machine port *number*.

Syntax:

INP(*number*)

Comments:

number represents a valid machine port number in the range 0-65535.

The INP function is one way in which a peripheral device can communicate with a GW-BASIC program.

INP is the complementary function to the OUT statement.

Examples:

```
100 A=INP(56)
```

Upon execution, variable A contains the value present on port 56. The number returned is in the range 0-255, decimal.

The equivalent to the above statement in assembly language is:

```
MOV DX,56  
IN AL,DX
```

INPUT Statement

Purpose:

To prepare the program for input from the terminal during program execution.

Syntax:

INPUT[;][*prompt string*;] *list of variables*

INPUT[;][*prompt string*,] *list of variables*

Comments:

prompt string is a request for data to be supplied during program execution.

list of variables contains the variable(s) that stores the data in the *prompt string*.

You must surround each data item in the *prompt string* by quotation marks, followed by a semicolon or comma and the name of the variable to which it is to be assigned. If you use more than one variable, you must separate data items with commas.

The data entered is assigned to the variable list. The number of data items supplied must be the same as the number of variables in the list.

The variable names in the list can be numeric or string variable names (including subscripted variables). The type of each data item input must agree with the type specified by the variable name.

The use of too many or too few data items, or the wrong type of values (for example, numeric instead of string), causes the message ?Redo from start to be displayed. No assignment of input values is made until an acceptable response is given.

You can use a comma instead of a semicolon after *prompt string* to suppress the question mark. For example, the following line prints the prompt with no question mark:

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

BASIC suppresses the carriage return if the prompt string is preceded by a semicolon. During program execution, data on that line is displayed, and data from the next PRINT statement is added to the line.

When GW-BASIC encounters an INPUT statement during program execution, it halts the program, displays the prompt string, and waits for the operator to type the requested data. Strings that input to an INPUT statement need not be surrounded by quotation marks unless they contain commas or leading or trailing blanks.

When you press `[ENTER]`, program execution continues.

INPUT and LINE INPUT statements have built-in PRINT statements. When GW-BASIC encounters an INPUT statement with a quoted string during program execution, the quoted string is printed automatically. (See the PRINT statement.)

The principal difference between the INPUT and LINE INPUT statements is that LINE INPUT accepts special characters (such as commas) within a string, without requiring quotes; the INPUT statement requires quotes.

Examples:

The following program finds the square of a number:

```
10 INPUT X
20 PRINT X "SQUARED IS" X ^ 2
30 END

RUN
?
```

If you type a number (5) in response to the question mark the screen displays:

```
5 SQUARED IS 25
```

The following program finds the area of a circle when the radius is known:

```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R ^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20

RUN
WHAT IS THE RADIUS? 7.4
THE AREA OF THE CIRCLE IS 171.9464
```

Note that Line 20 in the above example makes use of the built-in PRINT statement contained within INPUT.

INPUT# Statement

Purpose:

To read data items from a sequential file and assign them to program variables.

Syntax:

INPUT# *file number, variable list*

Comments:

file number is the number used when the file was opened for input.

variable list contains the variable names to assigned to items in the file.

The data items in the file appear in the same manner as they do if you type data on the keyboard in response to an INPUT statement.

The variable type must match the type specified by the variable name.

With INPUT#, no question mark is printed, as it is with INPUT.

Numeric Values: For numeric values, GW-BASIC ignores leading spaces and line feeds. It assumes the first character it encounters (not a space or line feed) is the start of a number. The number terminates on a space, carriage return, line feed, or comma.

Strings: If GW-BASIC is scanning the sequential data file for a string, it ignores leading spaces and line feeds.

If the first character is a quotation mark ("), the string consists of all characters read between the first quotation mark and the second. A quoted string cannot contain a quotation mark as a character. The second quotation mark always terminates the string.

If the first character of the string is not a quotation mark, the string terminates on a comma, carriage return, line feed, or after 255 characters have been read.

If the end of the file is reached when a numeric or string item is being input, the item is terminated.

INPUT# can also be used with random files.

INPUT\$ Function

Purpose:

To return a string of *number* characters read from the keyboard or from *file number*.

Syntax:

INPUT\$(*number*[,*[#] file number*])

Comments:

If you use the keyboard for input, characters do not appear on the screen. All control characters (except **CTRL** **BREAK**) pass through. This key sequence interrupts the execution of the INPUT\$ function.

The INPUT\$ function is preferred over the INPUT and LINE INPUT statements for reading communications files because all ASCII characters can be significant in communications. INPUT is the 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\$ returns *number* characters from the file number or keyboard.

For more information about communications, refer to Appendix E in the *Tandy GW-BASIC User's Guide*.

Examples:

The following example lists the contents of a sequential file in hexadecimal.

```
10 OPEN"1",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

In the following example, the program pauses, awaiting a keyboard entry of either P or S. Line 130 continues to loop back to Line 100 if the input is other than P or S.

```
.  
.   
.   
100 PRINT "TYPE P TO PROCEED OR S TO STOP"  
110 X$=INPUT$(1)  
120 IF X$="P" THEN 500  
130 IF X$="S" THEN 700 ELSE 100  
.   
.
```

INSTR Function

Purpose:

To search for the first occurrence of *string2* in *string1*, and return the position at which the string is found.

Syntax:

INSTR([*number*],*string1*,*string2*)

Comments:

Optional offset *number* sets the position for starting the search. The default value for *number* is 1.

number must be in the range 1-255. If *number* is out of range, an Illegal Function Call error is returned.

If *number* equals zero, GW-BASIC returns an Illegal argument in line number error.

INSTR returns 0 if:

- *number* > LEN(*string1*)
- *string1* is null
- *string2* cannot be found

If *string2* is null, INSTR returns *number*.

string1 and *string2* can be string variables, string expressions, or string literals.

Examples:

```
10 STRING1="ABCDEBXYZ"
20 STRING2="B"
30 PRINT INSTR(STRING1,STRING2);INSTR(4,STRING1,
  STRING2)
RUN
      2      6
```

The interpreter searches the string "ABCDFBXYZ" and finds the first occurrence of the character B at position 2 in the string. It then starts another search at position 4 (D) and finds the second match at position 6 (B). The last three characters are ignored, since all conditions set out in Line 30 were satisfied.

INT Function

Purpose:

To truncate an expression to a whole number.

Syntax:

`INT(number)`

Comments:

Negative numbers return the next lowest number.

The FIX and CINT functions also return integer values.

Examples:

```
PRINT INT(98.89)
          98
```

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

IOCTL Statement

Purpose:

To allow GW-BASIC to send a "control data" string to a character device driver any time after the driver has been opened.

Syntax:

IOCTL[#] *file number, string*

Comments:

file number is the file number open to the device driver.

string is a valid string expression containing characters that control the device.

IOCTL commands are generally 2 to 3 characters and are followed by an optional alphanumeric argument. An IOCTL string can be up to 255 bytes long, with commands within the string separated by semicolons.

Examples:

If a user had installed a driver to replace Lpt1, and that driver was able to set page length (the number of lines to print on a page before issuing a form feed), then the following lines would open the new Lpt1 driver and set the page length to 66 lines:

```
OPEN "LPT1:" FOR OUTPUT AS #1
IOCTL #1,"PL66"
```

The following statements open Lpt1 with an initial page length of 56 lines:

```
OPEN "\\DEV\\LPT1" FOR OUTPUT AS #1
IOCTL #1,"PL56"
```

IOCTL\$ Function

Purpose:

To allow GW-BASIC to read a "control data" string from an open character device driver.

Syntax:

IOCTL\$([#] *file number*)

Comments:

file number is the file number open to the device.

You generally use the IOCTL\$ function to get acknowledgement that an IOCTL statement succeeded or failed. You can also use it to get device information, such as device width after an IOCTL statement requests it.

Examples:

```
10 'GW is a possible command
20 'for get device width
30 OPEN "\\DEV\\MYLPT" AS#1
40 IOCTL#1,"GW"
50 'Save it in WID
60 WID=VAL(IOCTL$(#1))
```

KEY Statement

Purpose:

To allow you to rapidly enter as many as 15 characters into a program with one keystroke by redefining GW-BASIC special function keys.

Syntax:

```
KEY key number, string expression  
KEY n, CHR$(hexcode) + CHR$(scan code)  
KEY ON  
KEY OFF  
KEY LIST
```

Comments:

key number is the number of the key to be redefined. *key number* can be in the range 1-10 or 15-20.

string expression is the key assignment. You can use any valid string of 1 to 15 characters. If a string is longer than 15 characters, KEY assigns only the first 15. Constants must be enclosed in quotation marks.

scan code is the variable defining the key you want to trap. Appendix G in the *Tandy GW-BASIC User's Guide* lists the scan codes for the keyboard keys.

hexcode is the hexadecimal code assigned to the key as shown below:

Key	Hexcode
EXTENDED	&H80
CAPS LOCK	&H40
NUM LOCK	&H20
ALT	&H08
CTRL	&H04
SHIFT	&H01, &H02, &H03

Hexcodes can be added together, such as &H03, which is both shift keys.

Initially, GW-BASIC assigns the function keys the following special functions:

F1	LIST	F2	RUN†
F3	LOAD"	F4	SAVE"
F5	CONT†	F6	,"LPT1:†
F7	TRON†	F8	TROFF†
F9	KEY	F10	SCREEN 000†

Note: The † symbol indicates that you do not have to press ENTER after each of these keys has been pressed.

You can redefine any number of the function keys. When you press the key, the data assigned to it is sent to the program.

KEY *key number*,"string expression"

Assigns the string expression to the specified key.

KEY LIST

Lists all 10 key values on the screen. All 15 characters of each value are displayed.

KEY ON

Displays the first six characters of the key values on the 25th line of the screen. When the display width is set at 40, five of the 10 keys are displayed. When the width is set at 80, all 10 are displayed.

KEY OFF

Erases the key display from the 25th line, making that line available for program use. KEY OFF does not disable the function keys.

If the value for *key number* is not in the range 1-10 or 15-20, an illegal function call error occurs. The previous KEY assignment is retained.

Assigning a null string (length 0) disables the key as a function key.

When a function key is redefined, the INKEY\$ function returns one character of the assigned string per invocation. If the function key is disabled, INKEY\$ returns a string of two characters: the first is binary zero; the second is the key scan code.

Examples:

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

Displays a menu selected by the operator each time Key 1 is pressed.

```
10 KEY OFF
```

Turns off the key display.

```
10 DATA KEY1,KEY2,KEY3,KEY4,KEY5
20 FOR N=1 TO 5:READ SOFTKEYS$(N)
30 KEY N,SOFTKEYS$(I)
40 NEXT N
50 KEY ON
```

Displays new function keys on Line 25 of the screen.

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

Disables Function Key 1.

KEY(number) Statement

Purpose:

To initiate and terminate key capture in a GW-BASIC program.

Syntax:

```
KEY(number) ON  
KEY(number) OFF  
KEY(number) STOP
```

Comments:

number is a number from 1 to 20 that indicates which key is to be captured. Keys are numbered as follows:

Key Number	Key
1-10	Function keys F1 through F10
11	CURSOR UP
12	CURSOR LEFT
13	CURSOR RIGHT
14	CURSOR DOWN
15-20	Keys defined in the following format: KEY <i>number</i> ,CHR\$(<i>hexcode</i>) + CHR\$(<i>scan code</i>) (See the KEY statement):

Execute KEY(*number*) ON to activate keystroke capture from function keys or cursor control keys. When the KEY(*number*) ON statement is activated and enabled, GW-BASIC checks each new statement to see whether you are pressing the specified key. If so, it performs a GOSUB to the line number specified in the KEY(*number*) ON statement. A KEY(*number*) ON statement must precede a KEY(*number*) statement.

When KEY(*number*) OFF is executed, no key capture occurs and keystrokes are not retained. If KEY(*number*) STOP executes, no key capture occurs, but if a specified key is pressed, the keystroke is retained so that keystroke capture occurs when a KEY(*number*) ON is executed.

For further information on key trapping, see the ON KEY (*number*) statement.

KILL Command

Purpose:

To delete a file from a diskette.

Syntax:

KILL *filename*

Comments:

filename can be a program file, sequential file, or random-access data file.

Note: You must specify the filename's extension when using the KILL command. Remember that files saved in GW-BASIC are given the default extension .bas.

If a KILL command is given for a file that is currently open, a File already open error occurs.

Examples:

The following command deletes the GW-BASIC file Data, and makes the space available for reallocation to another file:

```
200 KILL "DATA1.BAS"
```

The following command deletes the GW-BASIC file Raining from the subdirectory Dogs:

```
KILL "CATS\DOGS\RAINING.BAS"
```

LEFT\$ Function

Purpose:

To return the specified *number* of characters from the leftmost portion of *string*.

Syntax:

LEFT\$(*string, number*)

Comments:

number must be in the range 0-255. If *number* is greater than the length of *string*, the entire *string* is returned. If *number* equals zero, the null string (length zero) is returned. (See the MID\$ and RIGHT\$ substring functions.)

Examples:

```
10 A$="BASIC"  
20 B$=LEFT$(A$,3)  
30 PRINT B$
```

Displays the leftmost three characters, BAS.

LEN Function

Purpose:

To return the number of characters in *string*.

Syntax:

LEN(*string*)

Comments:

Nonprinting characters and blanks are counted.

Examples:

string is any string expression.

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

Displays 16. Note that the comma and space are included in the character count of 16.

LET Statement

Purpose:

To assign the value of an *expression* to a *variable*.

Syntax:

[LET] *variable* = *expression*

Comments:

The word LET is optional. The equal sign is sufficient.

LET is seldom used. It is included in GW-BASIC to ensure compatibility with previous versions of GW-BASIC that require it.

When using LET, remember that the type of the *variable* and the type of the *expression* must match. If they do not, a Type mismatch error occurs.

Examples:

The following examples give downward compatibility with an older system. If this downward compatibility is not required, use the second example; it requires less memory.

```
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
```

```
.
.
.
```

LINE Statement

Purpose:

To draw lines and boxes on the screen.

Syntax:

LINE [(x1, y1)]-(x2, y2) [, [*attribute*][, b[f]] [, *style*]]

Comments:

x1, *y1* and *x2*, *y2* specify the end points of a line.

The resolution is determined by the screen mode set with the SCREEN statement.

attribute specifies the color or intensity of the displayed pixel. (See the COLOR and PALETTE statements.)

If you use the b option, GW-BASIC draws a box with the points *x1*, *y1* and *x2*, *y2* at opposite corners.

If you use the bf option, GW-BASIC draws a filled box (as ,b) and fills in the interior with points.

Note: If you omit *attribute*, you must use two commas before b or bf.

LINE supports the additional argument *style*. *style* is a 16-bit integer mask used when putting pixels on the screen. Using such a mask is called *line styling*.

Each time LINE stores a point on the screen, it uses the current circulating bit in *style*. If that bit is 0, LINE does not store the point. If the bit is a 1, a normal store is done. After each point, the next bit position in *style* is selected.

Because a 0 bit in *style* does not clear out the old contents, you might wish to draw a background line before a styled line, to force a known background.

Use *style* only with normal lines and boxes. It is invalid for filled boxes.

If you use bf with the *style* parameter, a Syntax error occurs.

If you use out-of-range values in the LINE statement, the coordinates that are out of range are not visible on the screen. This is called line clipping.

In the LINE syntax, the coordinate form STEP (*x offset*, *y offset*) is not shown. However, you can use this form for any coordinate.

In a LINE statement, if the relative form is used on the second coordinate, that coordinate is relative to the first coordinate.

After a LINE statement, the last referenced point is *x2*, *y2*.

The simplest form of LINE is:

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

This example draws a line from the last point referenced to the point *x2*,*y2* in the foreground color.

Examples:

```
SCREEN DISP D = LINE (0,100)-(639,100)
```

Draws a horizontal line across the middle of the screen in high resolution (Screen Mode 2).

```
SCREEN DISP D = LINE (160,0)-(160,199)
```

Draws a vertical line down the center of the screen in medium resolution (Screen Mode 1), or draws a one-quarter/three-quarter dividing line in high resolution (Screen Mode 2).

```
LINE (0,0)-(319,199)
```

Draws a diagonal line from the upper left corner to the lower right corner in medium resolution, or draws from the upper left corner to the center bottom of the screen in high resolution.

```
LINE (10,10)-(20,20),2
```

Draws a line in Color 2 if you previously selected medium resolution. (See the COLOR statement.)

Draws a line in Color 2 if you previously selected medium resolution.
(See the COLOR statement.)

```
10 CLS
20 LINE -(RND*319,RND*199),RND*4
30 GOTO 20
```

Draws lines forever, using the random attribute.

```
10 FOR X=0 TO 319
20 LINE (X,0)-(X,199),X AND 1
30 NEXT
```

Draws an alternating pattern: line on, line off.

```
10 CLS
20 LINE -(RND*639,RND*199),RND*2,BF
30 GOTO 20
```

Draws lines all over the screen.

```
LINE (0,0)-(100,175),,B
```

Draws a square in the upper left corner if you specified medium resolution.

```
LINE (0,0)-(100,175),,BF
```

Draws the same square and fills it in.

```
LINE (0,0)-(100,175),2,BF
```

Draws the same filled square in magenta if you specified medium resolution.

```
LINE (0,0)-(100,350),,B
```

Draws the same square if you specified high resolution.

400 SCREEN 1

410 LINE(160,100)-(160,199),,,&HCCCC

Draws a dotted line down the center of the screen in medium resolution.

220 SCREEN 2

230 LINE(300,100)-(400,50),,B,&HAAAA

Draws a rectangle with a dotted line in high resolution.

LINE (0,0)-(160,100),3,,&HFF00

Draws a dotted line from the upper left corner to the center of the screen.

LINE INPUT Statement

Purpose:

To input an entire line (up to 255 characters) from the keyboard into a *string variable*, ignoring delimiters.

Syntax:

LINE INPUT [;][*prompt string*; *string variable*

Comments:

prompt string is a string literal, displayed on the screen, that allows user input during program execution.

A question mark follows the printed prompt only if you include the question mark in the *prompt string*.

string variable accepts all input from the end of the prompt to a carriage return. Trailing blanks are ignored.

LINE INPUT is almost the same as the INPUT statement, except that it accepts special characters (such as commas) in operator input during program execution.

If a line feed/carriage return sequence (this order only) is encountered, both characters are input and echoed. Data input continues. If LINE INPUT is immediately followed by a semicolon, pressing **ENTER** does not move the cursor to the next line.

To terminate a LINE INPUT without entering one or more characters, press **CTRL** **BREAK**. GW-BASIC returns to command level and displays Ok.

Typing CONT causes execution to resume at the LINE INPUT line.

Examples:

100 LINE INPUT A\$

Pauses program execution at Line 100. All keyboard characters that the user types thereafter are input to string A\$ until you press one of the following keys or key sequences:

ENTER

CTRL M

CTRL C

CTRL BREAK

LINE INPUT# Statement

Purpose:

To read an entire line (up to 255 characters), without delimiters, from a sequential disk file to a *string variable*.

Syntax:

LINE INPUT# *file number, string variable*

Comments:

file number is the number used to open the file.

string variable is the variable name to which you want the line assigned.

LINE INPUT# reads all characters in the sequential file up to a carriage return. If a line feed/carriage return sequence (this order only), is encountered, it is input.

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a GW-BASIC program saved in ASCII mode is being read as data by another program.

Examples:

```
10 OPEN "O",1,"INFO"
20 LINE INPUT "CUSTOMER INFORMATION?";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"INFO"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
```

Displays:

CUSTOMER INFORMATION?

If you enter:

LINDA JONES 234,4 MEMPHIS

the program continues with:

LINDA JONES 234,4 MEMPHIS

LIST Command

Purpose:

To list all or part of the program currently in memory to the screen, a line printer, or a file.

Syntax:

```
LIST [startline][-endline][,device]  
LIST [startline-][,device]
```

Comments:

Use *startline-endline* to specify a range of lines to list. Both numbers must be valid line numbers in the range 0-65529.

If you omit the line range, the entire program is listed. If you include only *startline*-, GW-BASIC lists the specified line and all higher-numbered lines. If you include only *-endline*, GW-BASIC lists lines from the beginning of the program to the specified line.

You can substitute a period (.) for either line number to indicate the current line.

device specifies a file or device (LPT1: for line printer or SCRN: for screen) to which to list the lines. If you omit *device*, the lines are listed to the screen.

LIST "LPT1:" lists the program to the line printer. This statement is equivalent to LLIST.

You can interrupt any listing by pressing CTRL BREAK .

Examples:

LIST

Lists all lines in the program.

LIST -20

Lists Lines 1-20.

LIST 10-20

Lists Lines 10-20.

LIST 20-

Lists Line 20 and all higher-numbered lines.

LLIST Command

Purpose:

To list all or part of the program currently in memory to the line printer.

Syntax:

```
LLIST [startline][-endline]  
LLIST [startline-]
```

Comments:

GW-BASIC always returns to command level after an LLIST is executed. The line range options for LLIST are the same as for LIST.

Examples:

See the examples for the LIST statement.

LOAD Command

Purpose:

To load a file from diskette into memory.

Syntax:

LOAD *filename*[,r]

Comments:

filename is the filename used to save the file. If you omit the extension, GW-BASIC assumes the extension .bas.

Before loading the designated file, LOAD closes all open files and deletes all variables and program lines currently residing in memory.

The r option tells GW-BASIC to run the program after loading it and to keep open all open data files.

Using the r option also lets you chain several programs (or segments of the same program). Information can be passed between the programs using the disk data files.

Examples:

LOAD "STRTRK",R

Loads the file Strtrk.bas and runs it, retaining all open files and variables from a previous program intact.

LOC Function

Purpose:

To return the current position in the file.

Syntax:

LOC(*file number*)

Comments:

file number is the file number used to open the file.

When transmitting or receiving a file through a communication port, LOC returns the number of characters in the input buffer waiting to be read. The default size for the input buffer is 256 characters, but you can change the size by using /c: option when loading GW-BASIC.

If the buffer contains more than 255 characters, LOC returns only 255. As a string is limited to 255 characters, the limit on LOC alleviates the need to test for the amount of data before reading it into a string. If fewer than 255 characters remain in the buffer, then LOC returns the actual count.

With random disk files, LOC returns the number of the record just read or written with a GET or PUT statement.

With sequential files, LOC returns the number of 128-byte blocks read from, or written to, the file since it was opened. When the sequential file is opened for input, GW-BASIC initially reads the first sector of the file. In this case, the LOC function returns the character 1 before any input is allowed.

If the file was opened but no disk input/output was performed, LOC returns 0.

Examples:

```
200 IF LOC(1) > 50 THEN STOP
```

The program stops when 51 records have been read or written since the file was opened.

LOCATE Statement

Purpose:

To move the cursor to the specified position on the active screen. Optional parameters cause the *cursor* to blink on and off and define the *start* and *stop* raster lines for the cursor. A *raster line* is the vertical or horizontal distance between two adjacent, addressable points on your screen.

Syntax:

```
LOCATE [row][,[column]][,[cursor]][,[start] [, stop]]]
```

Comments:

row is the screen line number, a numeric expression in the range 1-25.

column is the screen column number, a numeric expression in the range 1-40 or 1-80, depending on screen width.

cursor is a Boolean value indicating whether the cursor is visible. Use zero to turn the cursor off (make it invisible). Use nonzero to turn it on.

start is the cursor starting scan line. It is a numeric expression in the range 0-31.

stop is the cursor stop scan line, a numeric expression in the range 0-31.

You can use LOCATE to move the cursor to a position at which you want subsequent PRINT statements to begin displaying characters. Optionally, you can use it to start the cursor blinking on or off or to change the size of the blinking cursor.

Any values entered outside of these ranges result in illegal function call errors. Previous values are retained.

To keep an existing LOCATE parameter the same, omit the parameter from the statement and replace the parameter with a comma. If the omitted parameter occurs at the end of the statement, do not type the comma.

If you include *start* and exclude *stop*, the *stop* value becomes the *start* value.

Examples:

10 LOCATE 1,1

Moves the cursor to the home position in the upper left corner.

20 LOCATE ,,1

Makes the cursor visible without moving it. Notice that the first two parameters are not used. A comma is inserted for each omitted parameter.

30 LOCATE,,,7

Retains the cursor's position, keeps its visibility the same, and causes the cursor to appear at the bottom of the character starting and ending on Scan Line 7.

40 LOCATE 5,1,1,0,7

Moves the cursor to Row 5, Column 1, and turns it on. The cursor covers an entire character cell, starting at Scan Line 0 and ending at Scan Line 7.

LOCK Statement

Purpose:

To restrict the access to all or part of a file opened by another process. This is used in a workgroup (multi-device) environment.

Syntax:

LOCK [#]*number* [, [*start record*] [TO *end record*]]

Comments:

number is the number the program originally gave the file when it was opened.

You can specify either one record (*start record*) or a range of records (*start record* TO *end record*) to lock. If you specify a range, *start record* must be less than or equal to *end record*.

The range of legal record numbers is 1 to $2^{32}-1$. The limit on record size is 32767 bytes.

If you omit *start record*, LOCK assumes Record 1.

If you omit *end record*, only the specified record is locked.

The following are valid variations of the LOCK statement:

LOCK # <i>number</i>	locks the entire file specified by <i>number</i> .
LOCK # <i>number</i> , <i>start record</i>	locks only the specified record.
LOCK # <i>number</i> , TO <i>end record</i>	locks Record 1 through the specified record.
LOCK # <i>number</i> , <i>start record</i> TO <i>end record</i>	locks all records in the range <i>start record</i> - <i>end record</i> .

With a random-access file, you can lock the entire open file or a range of records within the file. Thus, you can deny any other process (that has opened the file) access to certain records.

With a sequential access file that is open for input or output, you can lock only the entire file. If you specify a range of records, LOCK simply disregards the range.

Execute the LOCK statement on a file (or records in a file) before attempting to read or write to a file.

Caution: Always use the UNLOCK statement to unlock the locked file/records before closing the file. Failure to do so can jeopardize future access to that file in a workgroup environment.

Because you will probably want to lock files/records only for a short time, we recommend using LOCK within short-term paired LOCK/UNLOCK statements.

Examples:

The following sequence demonstrates how to use LOCK and UNLOCK statements:

```
LOCK #1, 1 TO 4
LOCK #1, 5 TO 8
UNLOCK #1, 1 TO 4
UNLOCK #1, 5 TO 8
```

The following example is illegal:

```
LOCK #1, 1 TO 4
LOCK #1, 5 TO 8
UNLOCK #1, 1 TO 8
```


LOF Function

Purpose:

To return the length (number of bytes) allocated to a file.

Syntax:

LOF(*file number*)

Comments:

file number is the number used to open the file.

With communications files, LOF returns the amount of free space in the input buffers.

Examples:

The following sequence gets the last record of the random-access file File.big, and assumes that the file was created with a default record length of 128 bytes:

```
10 OPEN "R",1,"FILE.BIG"  
20 GET #1,LOF(1)/128
```

```
.  
.   
.
```

LOG Function

Purpose:

To return the natural logarithm of *number*.

Syntax:

`LOG(number)`

Comments:

number must be a number greater than zero.

`LOG(number)` is calculated in single precision, unless the /d switch is used when GW-BASIC is executed.

Examples:

```
PRINT LOG(2)
```

Returns the value .6931471.

```
PRINT LOG(1)
```

Returns the value 0.

LPOS Function

Purpose:

To return the current position of the line printer print head in the line printer buffer.

Syntax:

LPOS(*number*)

Comments:

LPOS does not necessarily give the physical position of the print head.

number is a dummy argument.

If the printer has less than the 132 characters-per-line capability, it might issue internal line feeds and not inform the computer internal line printer buffer. If this happens, the value returned by LPOS might not be correct. LPOS simply counts the number of printable characters since the last line feed was issued.

Examples:

```
100 IF LPOS(X) > 60 THEN LPRINT CHR$(13)
```

Causes a carriage return after the 60th character is printed on a line.

LPRINT and LPRINT USING Statements

Purpose:

To print *data* at the line printer. LPRINT USING lets you print the *data* using a specified *format*.

Syntax:

```
LPRINT {data[,data,...]}  
LPRINT USING format; data[,data,...]
```

Comments:

data can be strings and/or numeric expressions or values. If you specify more than one data item in the statement, use the same separators described in PRINT.

format consists of one or more field specifier(s) or any alphanumeric character. *format* must be enclosed in quotation marks.

LPRINT and LPRINT USING are the same as PRINT and PRINT USING, except that the output goes to the line printer. For more information about string and numeric fields and the variables used in them, see the PRINT and PRINT USING statements.

LPRINT and LPRINT USING assume that your printer allows a maximum of 80 characters per line. If your printer can print more than 80 characters per line, you can reset the number of characters that you can print on each line. See the WIDTH statement for more information.

LSET Statement

Purpose:

To move *data* from memory to a random file buffer, then left-justify it in preparation for a PUT statement.

Syntax:

LSET *field name* = *data*

Comments:

field name is a string variable defined in a FIELD statement.

Before using LSET, you must use FIELD to set up buffer fields.

To convert numeric values to string values so that you can left-justify them, see MKI\$, MKD\$, and MKS\$.

If the field is larger than the variable to which it is going, the data is left-justified and the field is padded with blanks on the right. If the variable is larger than the field, characters are truncated on the right.

LSET can also be used with a nonfielded string variable to left-justify or right-justify a string in a given field.

Examples:

```
110 A$ = SPACE$(20)
120 RSET A$ = N$
```

Right-justifies the string N\$ in a 20-character field. This can be valuable for formatting printed output.

MERGE Command

Purpose:

To merge the lines from an ASCII program file into the program already in memory.

Syntax:

MERGE *filename*

Comments:

filename is a valid string expression containing the filename. If you omit the filename extension, GW-BASIC assumes the extension .bas.

GW-BASIC searches the disk for the named program file. If it finds the file, it merges that program's lines with the lines in memory. After the MERGE command, the merged program resides in memory, and GW-BASIC returns to the direct mode.

The program being merged must be in ASCII format (saved with the a option to the SAVE command). If it is not, a **Bad file mode** error occurs, and the program in memory remains unchanged.

If any line numbers in the disk file are the same as any in the memory program, the disk file lines replace the corresponding lines in memory.

Examples:

MERGE "SUBRTN"

Merges the file Subrtn.bas with the program currently in memory, provided Subrtn was previously saved in ASCII format. If any lines in Subrtn are numbered the same as any lines in memory, the corresponding memory lines are replaced by the lines from Subrtn.

MID\$ Function

Purpose:

To return a substring of a *string*.

Syntax:

MID\$(*string*, *start*[*length*])

Comments:

length is the number of characters to return. It must be in the range 0-255.

start specifies the position in the string from which to get the substring. It must be in the range 1-255.

If you omit *length*, or if there are fewer than that number of characters to the right of the *start* position, GW-BASIC returns the *start* character and all characters to the right of it.

If *start* is greater than the number of characters in *string*, MID\$ function returns a null string.

If *length* equals 0, the MID\$ function returns a null string.

If either *start* or *length* is out of range, an illegal function call error is returned.

For more information and examples, see the LEFT\$ and RIGHT\$ functions.

Examples:

```
10 A$="GOOD"  
20 B$="MORNING EVENING AFTERNOON"  
30 PRINT A$;MID$(B$,8,8)
```

Line 30 combines GOOD (the A\$ string) to EVENING (the string of eight characters that begins at Position 8 in B\$), and displays the result (GOOD EVENING).

MID\$ Statement

Purpose:

To replace a portion of *oldstring* with *newstring*.

Syntax:

`MID$(oldstring, start[,length]) = newstring`

Comments:

Both *start* and *length* are integer expressions.

oldstring and *newstring* are string expressions.

The characters in *oldstring*, beginning at the *start* position, are replaced by the characters in *newstring*.

length is an optional number specifying the number of characters from *newstring* that you want used in the replacement. If you omit *length*, the entire *newstring* is used.

Whether or not you include *length*, the replacement of characters never goes beyond the original length of *oldstring*.

Examples:

```
10 A$="KANSAS CITY, MO"  
20 MID$(A$,14)="KS"  
30 PRINT A$
```

Line 20 overwrites MO in the A\$ string with KS to display KANSAS CITY, KS.

MKDIR Command

Purpose:

To create a subdirectory.

Syntax:

MKDIR *pathname*

Comments:

pathname is a string expression not exceeding 63 characters, identifying the subdirectory to be created.

Examples:

MKDIR "C:SALES\JOHN"

Creates the subdirectory John within the directory named Sales.

MKI\$, MKS\$, MKD\$ Functions

Purpose:

To convert numeric values to string values.

Syntax:

MKI\$(integer expression)
MKS\$(single-precision expression)
MKD\$(double-precision expression)

Comments:

MKI\$ converts an integer to a two-byte string.

MKS\$ converts a single-precision number to a four-byte string.

MKD\$ converts a double-precision number to an eight-byte string.

Any numeric value placed in a random file buffer with an LSET or RSET statement must be converted to a string. (See CVI, CVS, CVD, for the complementary functions.)

MKI\$, MKS\$, and MKD\$ are different from STR\$ because they change the interpretations of the bytes, not the bytes themselves.

Examples:

```
90 AMT = (K + T)
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
```

.
.
.

NAME Command

Purpose:

To rename *old filename* as *new filename*.

Syntax:

NAME *old filename* AS *new filename*

Comments:

old filename must exist, and *new filename* must not exist. Otherwise, an error results.

After a NAME command, the file exists on the same disk, in the same disk location, with the new name.

Examples:

NAME "ACCTS" AS "LEDGER"

Renames the file Accts to Ledger. The file's contents and location on the diskette remain unchanged.

NEW Command

Purpose:

To delete the program currently in memory and clear all variables.

Syntax:

NEW

Comments:

Enter NEW at the command level to clear memory before entering a new program. GW-BASIC always returns to command level after a NEW is executed.

Examples:

NEW

Clears the current program from memory

```
980 PRINT "Do You Wish To Quit (Y/N)
990 ANS$=INKEY$: IF ANS$=""THEN 990
1000 IF ANS$="Y" THEN NEW
1010 IF ANS$="N" THEN 980
1020 GOTO 990
```

OCT\$ Function

Purpose:

To convert a decimal value, *number*, to an octal value.

Syntax:

OCT\$(*number*)

Comments:

number is rounded to an integer before OCT\$(*number*) is evaluated.

This statement converts a decimal value in the range -32768 to +65535 to an octal string expression.

Octal numbers are numbers to the base 8 rather than base 10 (decimal numbers).

See the HEX\$ function for hexadecimal conversion.

Examples:

```
10 PRINT OCT$(18)
```

Returns 22, the octal equivalent of decimal 18.

ON Statement

**ON COM(*n*), ON KEY(*n*), ON PEN, ON PLAY(*n*),
ON STRIG(*n*), and ON TIMER(*n*)**

Purpose:

To cause program execution to branch to the specified *line* when the specified *event* (such as input from the communications port, use of the light pen, use of joysticks, or keypress of function or cursor control keys) occurs.

Syntax:

ON *event* GOSUB *line*

Comments:

event can be COM, KEY, PEN, PLAY, STRIG, or TIMER.

line is the number of the program line to which execution branches. You can disable trapping for events by setting *line* to 0.

Once you set trap line numbers, you can control event trapping itself with the following syntax lines:

- | | |
|-------------------|---|
| <i>event</i> ON | If an event is on and you specify a nonzero line for the trap, GW-BASIC checks, before executing each new statement, to see whether the specified event has occurred. If it has, GW-BASIC performs a GOSUB to the line specified in the ON statement. |
| <i>event</i> OFF | If an event is off, no trapping takes place. Even if the event occurs, GW-BASIC does not remember it. |
| <i>event</i> STOP | If an event is stopped, no trapping can take place. However, if the event occurs, GW-BASIC remembers the event and an immediate trap takes place when you execute <i>event</i> ON. |

When a trap is made for a particular event, the trap automatically causes a stop on that event. Thus, recursive traps can never take place.

The return from the trap routine automatically performs an ON unless an explicit OFF was performed inside the trap routine.

The occurrence of an error trap automatically disables all trapping.

Trapping takes place only when GW-BASIC is executing a program.

The following are valid values for *event*:

COM(<i>n</i>)	<i>n</i> is the number of the COM <i>channel</i> (1 or 2).
KEY(<i>n</i>)	<i>n</i> is a function key number 1-20. Numbers 1-10 correspond to function keys F1-F10. Numbers 11-14 correspond to the cursor control keys as follows: 11 = Cursor Up 13 = Cursor Right 12 = Cursor Left 14 = Cursor Down Numbers 15-20 are user-defined keys.
PEN	Because there is only one light pen, you do not specify a number with PEN.
PLAY(<i>n</i>)	<i>n</i> is an integer expression in the range 1-32. Values that are outside this range result in illegal function call errors.
STRIG(<i>n</i>)	<i>n</i> is 0, 2, 4, or 6 as follows: 0 = Left joystick, Button 1 2 = Right joystick, Button 1 4 = Left joystick, Button 2 6 = Right joystick, Button 2
TIMER(<i>n</i>)	<i>n</i> is a numeric expression in the range 1-86400. A value outside this range results in an illegal function call error.

RETURN *line*

This optional form of RETURN is primarily intended for use with event trapping. Use it to return from the trapping routine to a specific line in the GW-BASIC program while still eliminating the GOSUB entry that the trap created.

Use the nonlocal RETURN with care. Any other GOSUB, WHILE, or FOR that is active at the time of the trap remains active.

If the trap comes out of a subroutine, any attempt to continue loops outside the subroutine results in a NEXT without FOR error.

Special Notes About Each Type of Trap

COM Trapping: Typically, the COM trap routine reads an entire message from the COM port before returning.

We do not recommend that you use the COM trap for single-character messages. At high baud rates, the overhead of trapping and reading for each individual character might cause the interrupt buffer for COM to overflow.

KEY Trapping: Trappable keys 15-20 are defined by the following statement:

KEY(*number*),CHR\$(*hex code*) + CHR\$(*scan code*)

number is an integer expression in the range 15-20, defining the key to be trapped.

hex code is the mask for these latched keys:

CAPS LOCK

NUM LOCK

ALT

CTRL

LEFT SHIFT

RIGHT SHIFT

scan code is the number identifying one of the 83 keys to trap. Refer to Appendix G in the *Tandy GW-BASIC User's Guide* for key scan codes.

To trap a key that is shifted, control-shifted, or alt-shifted, the appropriate bit in *hex code* must be set. *hex code* values are:

Mask	Hex Code	Indicates that
CAPS LOCK	&H40	CAPS LOCK is active
NUM LOCK	&H20	NUM LOCK is active
ALT	&H08	ALT is pressed
CTRL	&H04	CTRL is pressed
LEFT SHIFT	&H02	Left SHIFT is pressed
RIGHT SHIFT	&H01	Right SHIFT is pressed

To trap shifted keys, you can use the value &H01, &H02, or &H03. The left and right SHIFT keys are coupled when &H03 is used.

Refer to the KEY(*n*) statement for more information.

No type of trapping is activated when GW-BASIC is in direct mode. Function keys resume their standard expansion meanings during input.

A key that causes a trap is not available for examination with the INPUT or INKEY\$ statements, so you must use a different trap routine for each key that is to have a different function.

If [CTRL] [PRTSC] is trapped, the printer echo toggle is processed first. Defining [CTRL] [PRTSC] as a key trap does not prevent characters from echoing to the printer if [CTRL] [PRTSC] is pressed.

Function keys 1-14 are predefined. Therefore, setting Scan Code 59-68, 72, 75, 77, or 80 has no effect.

PLAY(*n*) Trapping: A PLAY event trap is issued only when playing background music (PLAY"MB..). GW-BASIC does not issue PLAY event music traps running in MUSIC foreground (default case or PLAY"MF..).

Choose conservative 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 your program.

The ON PLAY(*n*) statement causes an event trap when the background music queue goes from *n* to *n*-1 notes.

STRIG Trapping: Using STRIG(*n*) ON activates the interrupt routine that checks the status of the joysticks. Downstrokes that cause trapping do not set the STRIG(0), STRIG(2), STRIG(4), or STRIG(6) function.

TIMER(*n*) Trapping: An ON TIMER(*n*) event trapping statement is used with applications needing an internal timer. The trap occurs when *n* seconds have elapsed since the TIMER ON statement.

Examples:

The first example is a very simple terminal program:

```
10 REM "ON COM(n)" EXAMPLE
20 OPEN "COM1:9600,O,7" AS #1
30 ON COM(1) GOSUB 80
40 COM(1) ON
50 REM TRANSMIT CHARACTERS FROM KEYBOARD
60 A$ = INKEY$:IF A$ = "" THEN 50
70 PRINT #1,A$;:GOTO 50
80 REM DISPLAY RECEIVE CHARACTERS
90 ALL = LOC(1):IF ALL < 1 THEN RETURN
100 B$ = INPUT$(ALL,#1):PRINT B$;:RETURN
```

You can use the following program lines to prevent **CTRL** **BREAK** from stopping execution or to prevent a system reset during a program:

```
10 KEY 15,CHR$(4) + CHR$(70) REM Trap CTRL-BREAK
20 KEY 16,CHR$(12) + CHR$(83) REM Trap system reset
30 ON KEY(15) GOSUB 1000
40 ON KEY(16) GOSUB 2000
50 KEY(15) ON
60 KEY(16) ON
.
.
.
1000 PRINT "I'm sorry, I can't let you do that"
1010 RETURN
2000 ATTEMPS = ATTEMPS + 1
2010 ON ATTEMPS GOTO 2100,2200,2300,2400,2500
2100 PRINT "Mary had a little lamb":RETURN
2200 PRINT "Its fleece was white as snow":RETURN
2300 PRINT "And everywhere that Mary went":RETURN
2400 PRINT "The lamb was sure to go":RETURN
2500 KEY(16) OFF REM Pressing the key once more
2510 RETURN REM ends the program ...
```

The following program displays the time of day on Line 1 every minute.

```
10 ON TIMER(60) GOSUB 10000
20 TIMER ON
.
.
.
10000 OLDROW = CSRLIN REM Saves the current row
10010 OLDCOL = POS(0) REM Saves the current column
10020 LOCATE 1,1:PRINT TIME$
10030 LOCATE OLDROW,OLDCOL REM Restores row and
      column
10040 RETURN
```

ON ERROR GOTO Statement

Purpose:

To enable error trapping and specify the first line of the error-handling subroutine.

Syntax:

ON ERROR GOTO *line*

Comments:

When error-trapping is enabled, all errors that GW-BASIC detects, including direct mode errors such as syntax errors, cause GW-BASIC to branch to the starting *line* of the error subroutine.

GW-BASIC branches to the specified *line*. A RESUME statement causes GW-BASIC to continue execution after the error.

If *line* does not exist, an Undefined line error results.

To disable error trapping, execute the following statement:

ON ERROR GOTO 0

Subsequent errors display an error message and halt execution.

An ON ERROR GOTO 0 statement in an error-trapping subroutine causes GW-BASIC to stop and print the error message for the error that caused the trap. We recommend that all error-trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

If an error occurs during execution of an error-handling subroutine, the GW-BASIC error message is displayed and execution is terminated. Error trapping does not occur within the error-handling subroutine.

Examples:

```
10 ON ERROR GOTO 1000
.
.
.
1000 A = ERR:B = ERL
1010 PRINT A,B
1020 RESUME NEXT
```

Line 1010 displays the type and location of the error on the screen. (See the ERR and ERL variables.)

Line 1020 causes program execution to continue with the line following the error.

ON/GOSUB and ON/GOTO Statements

Purpose:

To branch to one of several specified *line numbers*, in accordance with the value returned when *expression* is evaluated.

Syntax:

ON *expression* GOTO *line numbers*
ON *expression* GOSUB *line numbers*

Comments:

The particular *line number* branched to depends on the value returned when the specified *expression* is evaluated. For example, if the value returned is 3, the third line number listed is the destination of the branch.

If the value returned is a noninteger, GW-BASIC automatically rounds the fractional portion to an integer before evaluating the number.

In the ON/GOSUB statement, each *line number* in the list must be the first line of a subroutine.

If the value of *expression* is zero or greater than the number of items in the list (but less than or equal to 255), GW-BASIC continues with the next executable statement.

If the value of *expression* is negative or greater than 255, an illegal function call error occurs.

Examples:

```
100 IF R < 1 OR R > 4 THEN PRINT "ERROR":END
```

Ends program execution if the integer value of R is less than 1 or greater than 4.

```
200 ON R GOTO 150,300,320,390
```

Causes the program to branch to Line 150 if R equals 1, to Line 300 if R equals 2, to Line 320 if R equals 3, or to Line 390 if R equals 4. Program execution continues from the line branched to.

OPEN Statement

Purpose:

To establish a path for input, output, or both, to a file or device.

Syntax:

OPEN *mode*,[#] *file number**filename*[, *record length*]

OPEN *filename* [FOR *mode*][*access*] AS [#] *file number*
[LEN=*record length*]

Comments:

filename is the name of the file.

The first syntax is for sequential files. In this syntax, *mode* is a string expression with one of the following characters:

Expression	Specifies
O	Sequential output mode
I	Sequential input mode
R	Random input/output mode
A	Position to end of file

The second syntax is for random-access files. In this syntax, the FOR *mode* clause determines your initial position in the file and the action to be taken if the file does not exist. If you omit the FOR *mode* clause, RANDOM is assumed.

The valid *modes* for random-access files are:

- | | |
|--------|---|
| INPUT | Positions to the beginning of the file. A File not found error is given if the file does not exist. |
| OUTPUT | Positions to the beginning of the file. If the file does not exist, GW-BASIC creates it. |
| APPEND | Positions to the end of the file. If the file does not exist, GW-BASIC creates it. |
| RANDOM | Specifies random input or output mode. With random I/O, you can read or write records at any position in the file. If the file is not found, GW-BASIC creates it. |

mode must be a string constant. Do not enclose *mode* in quotation marks.

access can READ, WRITE, or READ WRITE.

file number is a number between 1 and the maximum number of files allowed. The number associates an I/O buffer with a disk file or device. This association exists until you execute a CLOSE or CLOSE *file number* statement.

record length is an integer expression in the range 1-32767 that sets the record length to be used for random files. If omitted, *record length* defaults to 128 bytes.

When you use *record length* with sequential-access files, the default is 128 bytes and *record length* cannot exceed the value specified by the /s: switch.

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 and determines the mode of access used with the buffer.

More than one file can be opened for input or random access at one time with different file numbers. For example, the following statements are allowed:

```
OPEN "B:TEMP" FOR INPUT AS #1
OPEN "B:TEMP" FOR INPUT AS #2
```


However, a file can be opened only once for output or append. For example, the following statements are illegal:

```
OPEN "TEMP" FOR OUTPUT AS #1
OPEN "TEMP" FOR OUTPUT AS #2
```

Note: Be sure to close all files before removing diskettes from the disk drives. (See CLOSE and RESET.)

In place of a *filename*, you can specify any of the following devices in the *mode* listed:

A:,B:,C:...	Drives A, B, C, and so on
KYBD:	Keyboard (input only)
SCRN:	Screen (output only)
LPT1:	Line Printer 1 (output only)
LPT2:	Line Printer 2 (output only)
LPT3:	Line Printer 3 (output only)
COM1:	RS-232 Communications 1 (input, output, or random)
COM2:	RS-232 Communications 2 (input, output, or random)

Disk files allow all modes.

When you open a disk file for append, the file pointer is initially positioned to the end of the file. The record number is set to the last record (LOF(*number*)/128). You can then use PRINT, WRITE, or PUT to extend the file. The program can position the pointer elsewhere in the file, using a GET statement. If this is done, the mode changes to RANDOM and the pointer moves to the record indicated.

Once the pointer is moved from the end of the file, it can be returned to the end using the statement:

```
GET #number, LOF(number)/record length
```

Any value outside the given ranges results in an Illegal function call error. The file is not opened.

Attempts to write to a file that is open for input only, result in Bad file mode errors.

Attempts to write to a file that is opened for output only, result in Bad file mode errors.

Opening a file for output or append fails if the file is already open in any other mode.

Because you can refer to one file by different paths, it is nearly impossible for GW-BASIC to know that it is the same file simply by looking at the path. For this reason, GW-BASIC does not let you open for output or append two files that have the same name and that exist on the same disk, even if their paths are different. For example, if Mary is your working directory, the following statements all refer to the same file:

```
OPEN "REPORT"  
OPEN "\SALES\MARY\REPORT"  
OPEN "..\MARY\REPORT"  
OPEN "..\..\MARY\REPORT"
```

Any filename can be open under more than one file number at a time. Because each file number has a different buffer, several records from the same file can be kept in memory for quick access. This allows different modes to be used for different purposes or, for program clarity, different file numbers to be used for different modes of access.

If you use the `LEN=record length` option, *record length* cannot exceed the record length set by /s: when you loaded GW-BASIC.

In a network environment, the use of the OPEN statement is based on two different sets of circumstances:

- When you want to share devices for specific purposes by using the OPEN statement to restrict file access to specific *modes*, such as: INPUT, OUTPUT, APPEND, and the default (RANDOM).
- When you restrict files by the implementation of an OPEN statement that allows a process to specify locking to the successfully opened file. The locking determines a guaranteed exclusivity range on that file by the process while the OPEN statement is in effect.

lock can be:

SHARED	“deny none” mode. Any process on any machine can read from or write to the file. However, the default mode is not allowed by any of the modes, including SHARED.
LOCK READ	“deny read” mode. Only the current process can read the file. (The system grants LOCK READ only if no other process already has LOCK READ access to the file.) Another process cannot open the file, if it is currently open in default mode or with a read access.
LOCK WRITE	“deny write” mode. Only the current process can write to the file. (The system grants LOCK WRITE access only if no other process already has LOCK WRITE.) An attempt by any other process to open the file is unsuccessful if the file has been opened in default mode or with a write access by another process.
LOCK READ WRITE	“deny all,” or “exclusive,” mode. Only the current process can read from or write to the file. (The system grants LOCK READ WRITE only if no other process already has LOCK READ, LOCK WRITE, or LOCK READ WRITE access to the file.)
default	“compatibility” mode, where the compatibility with other GW-BASICs is understood. No access is specified. The file can be opened any number of times by a process, provided that the file is not opened by another process. Other processes cannot access the file while it is open under default access. Therefore, it is functionally exclusive.

If an attempt is made to open a file already accessed by another process, a Permission Denied error results. For example, this error occurs if a process tries to use OPEN SHARED with a file on which another process already has OPEN LOCK READ WRITE.

If an OPEN statement for a device fails because the *mode* is incompatible with network-installed sharing access, GW-BASIC generates a **Path/File Access Error**. This error occurs, for example, if a process attempts to open for output on a directory that has been shared for read only.

For more information about using files in a networking environment, see the LOCK and UNLOCK statements.

Examples:

```
10 OPEN "I",2,"INVEN"
```

Opens File 2, Inven, for sequential input.

OPEN "COM Statement

Purpose:

To allocate a buffer to support RS-232 asynchronous communications with other computers and peripheral devices in the same manner as OPEN for disk files.

Syntax:

```
OPEN "COM[channel]:[speed][, parity][,data], stop][,rs][,cs[n]]  
[,ds[n]][,cd[n]][,lf] [, pe]" AS [#]file number [LEN = number]
```

Comments:

channel can be 1 or 2 to select the communications channel to be opened.

speed is an integer specifying the transmit and receive rate in bits per second (BPS). Valid speeds are 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600. The default is 300 bps.

parity is a one-character literal specifying the parity to be used when the data is transmitted and received. Valid characters are:

S	SPACE. Parity bit always transmitted and received as space (0 bit).
M	MARK. Parity bit always transmitted and received as mark (1 bit).
O	ODD. Odd transmit parity; odd receive parity checking.
E	EVEN. Even transmit parity; even receive parity checking.
N	NONE. No transmit parity. No receive parity checking.

If you omit *parity*, GW-BASIC assumes E (EVEN).

data is an integer specifying the number of transmit and receive data bits. Valid values are 4, 5, 6, 7, and 8. The default is 7 bits.

Note: Specifying four data bits with no parity is illegal, as is specifying eight data bits **with** parity.

stop is a literal integer expression returning a valid file number. Valid values for the number of stop bits are 1 or 2. If you omit *stop*, 75 and 110 bps transmit two stop bits, and all other speeds transmit one stop bit.

file number is a number between 1 and the maximum number of files allowed. A communications device can be opened to only one *file number* at a time.

file number is associated with the file as long as the file is open and is used to refer other COM I/O statements to the file.

Coding errors within the filename string result in Bad file name errors. No indication is given as to which parameters are in error.

number is the maximum number of bytes that can be read from the communications buffer when using the GET or PUT default of 128 bytes.

A Device timeout error occurs if "data set ready" (DSR) is not detected.

The rs, cs, ds, dc, lf, and pe options affect the line signals as follows:

Option	Function
RS	suppresses RTS (request to send)
CS[n]	controls CTS (clear to send)
DS[n]	controls DSR (data set ready)
CD[n]	controls CD (carrier detect)
LF	sends a line feed at each return
PE	enables parity checking

n is the number of milliseconds to wait (0-65535) for that signal before a device timeout error occurs. The defaults are: 1000 for cs, 1000 for ds, and 0 for cd. If you omit *n*, timeout is set to 0.

See Appendix E in the *Tandy GW-BASIC User's Guide* for more information about communications.

Examples:

10 OPEN "COM1:" AS 1

Opens File 1 for communication, using all defaults (300 bps, even parity, seven data bits, and one stop bit).

20 OPEN "COM1:2400" AS #2

Opens File 2 for communication at a rate of 2400 bps with even parity and seven data bits.

10 OPEN "COM1:1200,N,8" AS #1

Opens File 1 for asynchronous I/O at 1200 bps. No parity is to be produced or checked.

OPTION BASE Statement

Purpose:

To declare the minimum *value* for array subscripts.

Syntax:

OPTION BASE *value*

Comments:

value can be 1 or 0. The default is 0.

If the statement OPTION BASE 1 is executed, the lowest value an array subscript can have is 1.

OPTION BASE gives an error only if you change the base value. This allows chained programs to have OPTION BASE statements as long as *value* is not changed from its initial setting.

Note: You must code the OPTION BASE statement before you can define or use any arrays (using the DIM statement). If you try to change the *value* after any arrays are in use, an error results.

OUT Statement

Purpose:

To send a *data byte* to a machine output *port*.

Syntax:

OUT *port,data byte*

Comments:

port is an integer in the range 0-65535.

data byte is an integer in the range 0-255.

OUT is complementary to the INP function.

Examples:

```
100 OUT 12345,225
```

Outputs the decimal value 225 to Port 12345. In assembly language, this is equivalent to:

```
MOV DX,12345
MOV AL,255
OUT DX,AL
```

PAINT Statement

Purpose:

To fill in a graphics figure with the selected attribute.

Syntax:

PAINT (x,y)[, *paint attribute*[, *border attribute*][, *background attribute*]]

Comments:

The PAINT statement fills an arbitrary graphics figure of the specified *border attribute* with the specified *paint attribute*.

paint attribute can be a numeric formula that specifies a valid color, or it can be a string formula that specifies tiling, as described in the following “Paint Tiling” section. If you omit *paint attribute*, the standard foreground attribute (3 or 1) is used. See the COLOR and PALETTE statements for more information.

If not specified, *border attribute* defaults to *paint attribute*. PAINT must start on a nonborder point. Otherwise, it has no effect.

PAINT can fill any figure, but painting jagged edges or very complex figures can cause an Out of memory error. The CLEAR statement can be used to increase the amount of stack space available.

If you specify points beyond the screen’s limits, the points are not plotted. No error occurs.

Paint Tiling: Paint tiling is similar to line styling. Like LINE, PAINT looks at a tiling mask each time a point is put down on the screen.

If *paint attribute* is a string formula, then tiling is performed.

The tile mask is always eight bits wide and can be 1-64 bytes long. Each byte in the tile string masks eight bits along the x-axis when putting down points. Each byte of the tile string is rotated as required to align along the y-axis, such that:

tile byte mask = $y \text{ MOD } \textit{tile length}$

where *y* is the position of the graphics cursor on the y-axis.

tile length is the length, in bytes, of the tile string you define (1-64 bytes).

This is done so that the tile pattern is replicated uniformly over the entire screen (as if PAINT (0,0).. were used).

x Increases →

x,y	Bit of Tile Byte								
	8	7	6	5	4	3	2	1	
0,0	●	●	●	●	●	●	●	●	Tile Byte 1
0,1	●	●	●	●	●	●	●	●	Tile Byte 2
0,2	●	●	●	●	●	●	●	●	Tile Byte 3
0,3	●	●	●	●	●	●	●	●	Tile Byte 4
0,4	●	●	●	●	●	●	●	●	Tile Byte 5.
.
.
.
0,63	●	●	●	●	●	●	●	●	Tile Byte 64

In low resolution graphics (320 x 200), the tile mask is eight bits wide and can be a maximum of 64 bytes long.

In high resolution graphics (640 x 200), one bit of tile mask equals one point on the screen. Therefore, GW-BASIC sets each position in the tile mask with the bit value of 1. You can paint the screen with X's using the following statement:

```
PAINT (320,100),CHR$(&H81) + CHR$(&H42) + CHR$(&H24) +  
CHR$(&H18) + CHR$(&H18) + CHR$(&H24) + CHR$(&H81)
```

This appears on the screen as:

x Increases →

x,y	8	7	6	5	4	3	2	1		
0,0	•							•	CHR\$(&H81)	Tile Byte 1
0,1		•					•		CHR\$(&H42)	Tile Byte 2
0,2			•			•			CHR\$(&H24)	Tile Byte 3
0,3				•	•				CHR\$(&H18)	Tile Byte 4
0,4				•	•				CHR\$(&H18)	Tile Byte 5
0,5			•			•			CHR\$(&H24)	Tile Byte 6
0,6		•					•		CHR\$(&H42)	Tile Byte 7
0,7	•							•	CHR\$(&H81)	Tile Byte 8

In medium resolution (Screen Mode 1), there are two bits per pixel. (See the SCREEN statement.) Because of this, each byte of the tile pattern describes only four pixels. In this case, every two bits of the tile byte describes one of the four possible colors associated with each of the four pixels to be put down.

The following chart shows the values for the given colors. Color 0 is the set background color.

Palette 0	Palette 1	Binary Value
green	cyan	01
red	magenta	10
brown	high-intensity white	11

background attribute specifies the background tile pattern or color byte to skip when checking for boundary termination. *background attribute* is a string formula returning one character. When omitted, it defaults to CHR\$(0).

Occasionally, you might want to paint tile over an already painted area that is the same color as two consecutive lines in the tile pattern. PAINT quits when it encounters two consecutive lines of the same color as the point being set. (The point is surrounded.) It is not possible to draw alternating blue and red lines on a red background without *background attribute*. PAINT stops as soon as the first red pixel is drawn. By specifying red (CHR\$(&HAA)) as the background attribute, you can draw the red line over the red background.

Specifying more than two consecutive bytes in the tile string that match the *background attribute* results in an illegal function call error.

Examples:

```
10 CLS
20 SCREEN 1
30 LINE (0,0)-(100,150),2,B
40 PAINT (50,50),1,2
50 LOCATE 20,1
```

Line 40 fills the box drawn in Line 30 with Color 1.

PALETTE, PALETTE USING Statements

Purpose:

To change one or more of the colors in the palette.

Syntax:

```
PALETTE [attribute,color]  
PALETTE USING array (index)
```

Comments:

The PALETTE statements work **only** for systems equipped with the an Enhanced Graphics Adapter (EGA).

A GW-BASIC palette contains a set of colors, with each color specified by an *attribute*. Each *attribute* is paired with a display *color*. The display *color* determines the actual color on the screen. It depends on both the screen mode setting and the type of monitor.

PALETTE with no arguments sets the palette to a known initial setting. This setting is the same as the setting when colors are first initialized.

If you specify arguments, *color* is displayed whenever *attribute* is specified in any statement that specifies a color. Any color changes on the screen occur immediately. Note that when graphics statements use color arguments, they are actually referring to attributes and not actual colors. PALETTE pairs attributes with actual colors.

For example, assume that the current palette consists of Colors 0, 1, 2, and 3. The following DRAW statement

```
DRAW "C3L100"
```

Selects Attribute 3, and draws a line of 100 pixels using the color associated with the Attribute 3, in this case, also 3.

If you execute this statement

PALETTE 3,2

Then, the color associated with Attribute 3 is changed to Color 2. All text or graphics currently displayed using Attribute 3 instantaneously change to Color 2. All text or graphics subsequently displayed with Attribute 3 will also be displayed in Color 2. The new palette of colors will contain Colors 0, 1, 2, and 2.

With the USING option, all entries in the palette can be modified with PALETTE statement. *array* is the name of an integer array. *index* specifies the index of the first array element in *array* to use in setting your palette. Each attribute in the palette is assigned a corresponding color from this array.

The array must be dimensioned large enough to set all the palette entries after *index*. For example, if you are assigning colors to all 16 attributes, and the index of the first array element given in your PALETTE USING statement is 5, then *array* must be dimensioned to hold at least 20 elements (since the number of elements from 5 to 20, inclusive, is 16):

```
DIM PAL%(20)
```

```
.  
. .  
. .  
. .
```

```
PALETTE USING PAL%(5)
```

If the *color* argument in an array entry is -1, the mapping for the associated attribute is not changed. All other negative numbers are illegal values for *color*.

You can use the *color* argument in the COLOR statement to set the default text color. (Remember that *color* arguments in other GW-BASIC statements are actually what are called *attributes* in this discussion.) This *color* argument specifies the way that text characters appear on the screen. Under a common initial palette setting, points colored with the *attribute* 0 appear as black on the screen. Using the PALETTE statement, you can, for example, change the mapping of *attribute* 0 from black to white.

Remember that a PALETTE statement executed without any parameters assigns all *attributes* their default *colors*.

The following table lists *attribute* and *color* ranges for various monitor types and screen modes.

SCREEN Mode	Monitor Attached	Adapter	Attribute Range	Color Range
0	Monochrome	MTDA	NA	NA
	Monochrome	EGA	0-15	0-2
	Color/CGA	NA	0-31 ^d	
	Color/Enhanced ^a	EGA	0-31 ^d	0-15
1	Color/CGA	NA	0-3	
	Color/Enhanced ^a	EGA	0-3	0-15
2	Color/CGA	NA	0-1	
	Color/Enhanced ^a	EGA	0-1	0-15
7	Color/Enhanced ^a	EGA	0-15	0-15
8	Color/Enhanced ^a	EGA _b	0-15	0-15
9	Enhanced ^a	EGA _b	0-3	0-15
	Enhanced ^a	EGA ^c	0-15	0-63
10	Monochrome	EGA	0-3	0-8

^a EGM monitor

^b With 64K of EGA memory.

^c With greater than 64K of EGA memory.

^d Attributes 16-31 refer to blinking versions of Colors 0-15.

NA = Not Applicable.

CGA = Color Graphics Adapter.

EGA = Enhanced Graphics Adapter.

MTPA = Monochrome Text Display Adapter.

SCREEN Color and Attribute Ranges

See the SCREEN statement for the colors available for various SCREEN mode, monitor, and graphics adapter combinations.

Examples:

PALETTE 0,2

Changes all points colored with Attribute 0 to Color 2.

PALETTE 0,-1

Does not modify the palette.

PALETTE USING A%(0)

Changes each palette entry. Since the array is initialized to zero when it is first declared, all attributes are now mapped to display Color 0. The screen will now appear as one color. However, it will still be possible to execute GW-BASIC statements.

PALETTE

Sets each palette entry to its appropriate initial display color. Actual initial colors depend on your screen/hardware configuration.

PCOPY Command

Purpose:

To copy one screen page to another in all screen modes.

Syntax:

PCOPY *source page, target page*

Comments:

source page is an integer expression in the range 0-*n*, where *n* is determined by the current video memory size and the size per page for the current screen mode.

target page has the same requirements as *source page*.

For more information, see CLEAR and SCREEN.

Examples:

PCOPY 1,2

Copies the contents of Page 1 to Page 2.

PEEK Function

Purpose:

To read a byte from a specified *memory location*.

Syntax:

PEEK(*memory location*)

Comments:

The value returned is an integer in the range 0-255.

memory location must be in the range 0-65535.

The DEF SEG statement last executed determines the absolute address that is peeked into.

PEEK is the complementary function of the POKE statement.

Examples:

```
10 A=PEEK(&H5A00)
```

Causes GW-BASIC to return the value of the byte stored in the assigned hex offset memory location 5A00 (23040 decimal) and store it in the variable A.

PEN Function

Purpose:

To read the light pen's coordinates.

Syntax:

$x = P(\text{number})$

Comments:

You must execute a PEN ON statment before executing the PEN function. If you do not, an illegal function call error occurs.

x is the numeric variable receiving the PEN value.

number is an integer in the range 0-9 that tells GW-BASIC what to return. Values 0-5 return x, y coordinates corresponding to the current screen mode. Values 6-9 return the character row or column position.

number can be:

- | | |
|---|--|
| 0 | Returns -1 if the pen button has been pressed since the last poll. Returns 0 if not. |
| 1 | Returns the x-coordinate (horizontal) at which the pen was last activated. The range is 0-319 for medium resolution and 0-639 for high resolution. |
| 2 | Returns the y-coordinate (vertical) at which the pen was last activated. The range is 0-199. |
| 3 | Returns -1 if the pen button is being pressed. Returns 0 if the button is up. |
| 4 | Returns the last known valid x-coordinate. The range is 0-319 for medium resolution and 0-639 for high resolution. |
| 5 | Returns the last known valid y-coordinate. The range is 0-199. |
| 6 | Returns the character row position at which the pen was last activated. The range is 1-24. |
| 7 | Returns the character column position at which the pen was last activated. The range is 1-40 or 1-80, depending on the screen width. |

- 8 Returns the last known valid character row. The range is 1-24.
- 9 Returns the last known valid character column position. The range is 1-40 or 1-80, depending on the screen width.

When the pen is in the border area of the screen, the values returned are inaccurate.

Examples:

```
50 PEN ON
60 FOR I = 1 to 500
70 X = PEN(0):X1 = PEN(3)
80 Print X,X1
90 NEXT
100 PEN OFF
```

Returns values indicating whether the pen button has been pressed since the last poll and whether it is currently being pressed.

PEN Statement

Purpose:

To read the light pen.

Syntax:

PEN ON
PEN OFF
PEN STOP

Comments:

PEN ON enables the PEN read function.

PEN OFF disables the PEN read function.

PEN STOP disables trapping. It remembers the event so that immediate trapping occurs when PEN ON is executed.

To speed execution, turn off the pen for programs that don't use it.

PLAY Statement

Purpose:

To play the music specified by *string*.

Syntax:

PLAY *string*

Comments:

string is a string expression (surrounded by quotation marks) consisting of one or more of the following music commands:

A-G The letters A-G are musical notes. An optional number sign (#) or plus sign (+) following a letter produces a sharp. A minus sign produces a flat. You can only specify sharp or flat notes that correspond to the black keys on a piano. The letters A, C, D, F, and G can be followed by a plus sign because they are followed by black keys on a piano. The letters A, B, D, E, and G can be followed by a minus sign because they are preceded by black keys on a piano.

L(*n*) Sets the length of the notes that follow. *n* can be in the range 1-64. Some common lengths are:

- | | |
|----|-----------------------------|
| 1 | indicates a whole note. |
| 2 | indicates a half note. |
| 4 | indicates a quarter note. |
| 8 | indicates an eighth note. |
| 16 | indicates a sixteenth note. |

To change the length for only one note, place *n* immediately after the note, omitting the L. For example, A16 is equivalent to L16A.

O(*n*) Sets the current octave (0-6). Each octave starts with C and ends with B. Octave 3 starts with middle C. The default is Octave 4.

P*n* Plays the note *n* in the next higher octave.

- N(*n*)** Plays note *n*. This option provides an alternative to specifying the letter and octave of the note. Here you specify the note by number (1-84). (There are 84 notes in the seven octaves.) Specifying 0 causes a rest.
- P(*n*)** Causes a pause. *n* can be in the range 1-64. It has the same meaning as *n* with the L option.
- T(*n*)** Tempo. This option sets the number of quarter notes in a minute. *n* can be in the range 32-255. The default is 120, a moderate tempo.
- .** Plays the note as a dotted note. GW-BASIC plays the note 3/2 as long as the period determined by L (length) times T (tempo). If you use multiple periods after a note, the playing time is scaled accordingly. For example, A with one period (A.) causes the note A to play one-and-a-half times the playing time determined by L (length of the note) times T (the tempo). Two periods placed after A (A..) cause the note to play 9/4 times as long. An A with three periods (A...) plays 27/8 as long. Periods can also be used after a P to increase the length of the pause as described above.
- MF** Music foreground. GW-BASIC runs the PLAY and SOUND statements in the foreground. This means that each note or sound starts only after the previous note or sound finishes. If you omit MF and MB, GW-BASIC assumes MF.
- MB** Music background. GW-BASIC runs the PLAY and SOUND statements in the background. This means that each note or sound is placed in a buffer, allowing the GW-BASIC program to continue execution while music plays in the background. As many as 32 notes and/or rests can play in the background at one time.
- MN** Music normal. Each note plays seven-eighths of the duration set by L (length).
- ML** Music legato. Each note plays the full duration set by L.
- MS** Music staccato. Each note plays three-quarters of the duration set by L.

Xstring; Executes a substring, where *string* is a variable assigned to a string of PLAY commands. The X command lets you execute a second substring from a string, much like GOSUB. You can have one string execute another, which executes a third, and so on. *string* is a string variable in your program that contains the substring you want to execute. *string* can contain an X command to execute another substring. The semicolon after the string name is required.

Because of the slow clock interrupt rate, some notes might not play at higher tempos, for example, 1.64 at T255. These note/tempo combinations must be determined through experimentation.

>n A greater-than symbol preceding the note *n* plays the note in the next higher octave.

<n A less-than symbol preceding the note *n* plays the note in the next lower octave.

Note: Numeric arguments follow the same syntax described under the DRAW statement.

n as an argument can be a constant, or it can be a variable with an equal sign (= *variable*) in front of it. A semicolon is required after the variable and also after the variable in *Xstring*.

Examples:

```
10 PLAY "C4F .C8F8.C16F8.G16A2F2"  
20 INPUT "CAN YOU NAME THAT TUNE ";A$  
40 IF A$ = "THE EYES OF TEXAS" THEN GOTO 50 ELSE PRINT  
   "TRY AGAIN":GOTO 10  
50 PRINT "THAT'S RIGHT!"
```

PLAY Function

Purpose:

To return the number of notes currently in the background music queue.

Syntax:

`PLAY(number)`

Comments:

number is a dummy argument. It can be any value.

The PLAY function returns 0 when in music foreground mode.

The maximum number of notes returned is 32.

Examples:

```
10 ' when 4 notes are left in
20 ' queue play another tune
30 PLAY "MBABCDABCDABCD"
40 IF PLAY (0) = 4 THEN 200
.
.
.
200 PLAY "MBCDEFCDEF"
```

PMAP Function (Graphics)

Purpose:

To return the physical or world coordinate for a specified coordinate.

Syntax:

number = PMAP (*expression*, *action*)

Comments:

This function is valid for graphics modes only.

number is the physical coordinate of the point that is to be mapped.

expression is an x- or y-coordinate represented by a numeric variable or expression. If *expression* is a physical coordinate, it must be within the limits of the screen. If *expression* is a world coordinate, it can be any single-precision floating point value.

action is one of the following:

- | | |
|---|--|
| 0 | maps logical expressions to physical x |
| 1 | maps logical expressions to physical y |
| 2 | maps physical expressions to logical x |
| 3 | maps physical expressions to logical y |

Use PMAP with WINDOW and VIEW to translate coordinates.

POINT Function

Purpose:

To return the color or attribute value of a point on the screen, or to return the current physical or world (logical) coordinates.

Syntax:

`POINT(x,y)`
`POINT(action)`

Comments:

`POINT (x, y)` lets you examine the color or attribute value of point *x, y*. *x* is the horizontal coordinate of the point. *y* is the vertical coordinate.

If you specify a point that is out of range, GW-BASIC returns a -1.

To retrieve the current graphics coordinates, specify only *x* or *y*.

See the `COLOR` and `PALETTE` statements for valid color and attribute values.

`POINT (action)` lets you return the current physical or world (logical) coordinates. *action* is one of the following:

- 0 Returns the current physical x-coordinate (horizontal).
- 1 Returns the current physical y-coordinate (vertical).
- 2 Returns the current world x-coordinate if `WINDOW` is active. Otherwise, returns the current physical x-coordinate, as in 0 above.
- 3 Returns the current world y-coordinate if `WINDOW` is active. Otherwise, returns the current physical y-coordinate, as in 1 above.

Examples:

```
10 SCREEN 1
20 FOR C=0 TO 3
30 PSET (10,10),C
40 IF POINT(10,10) < > C THEN PRINT "BROKEN BASIC!"
50 NEXT C
```

The following inverts the current state of a point:

```
10 SCREEN 2
20 IF POINT(1,1) < > 0 THEN PRESET(1,1) ELSE PSET(1,1)
```

The following also inverts a point.

```
20 PSET (1,1),1-POINT(1,1)
```

POKE Statement

Purpose:

To write (poke) *data byte* into *memory location*.

Syntax:

POKE *memory location*, *data byte*

Comments:

Both *memory location* and *data byte* must be integers.

memory location is the offset address of the memory location to be poked. The DEF SEG statement last executed determines the address. GW-BASIC does not check any offsets that are specified. *memory location* must be in the range 0-65535.

data byte must be in the range 0-255.

POKE is the complementary function of PEEK. The argument to PEEK is an address from which a byte is to be read.

POKE and PEEK can increase data storage efficiency and help in loading assembly language subroutines and passing arguments and results to and from assembly language subroutines.

Examples:

20 POKE &H5A00,&HFF

Places the decimal value 255 (&HFF) into the hex offset location (23040 decimal). See the PEEK function example.

POS Function

Purpose:

To return the current column position of the cursor.

Syntax:

POS(*number*)

Comments:

The leftmost position is 1.

number is a dummy argument.

Examples:

```
10 CLS
20 WIDTH 80
30 A$ = INKEY$:IF A$ = "" THEN GOTO 30 ELSE PRINT A$;
40 IF POS(X) > 10 THEN PRINT CHR$(13);
50 GOTO 30
```

Causes a carriage return after the tenth character is printed on each line of the screen.

PSET Statement

Purpose:

To display (PSET) a point at a specified location in graphics mode. To clear a point, see PRESET.

Syntax:

PSET [STEP] (*x*, *y*)[*color*]

Comments:

x is the horizontal coordinate of the point. *y* is the vertical coordinate.

color is the color of the point.

Specify absolute coordinates, or use the STEP option to specify relative coordinates. STEP indicates that *x* and *y* are offsets relative to the last point referenced. For example:

STEP(10,10)

Coordinate values can be beyond the edge of the screen. However, values outside the integer range -32768 to 32767 cause an **Overflow** error.

The upper left corner is always (0,0). The lower left corner is (0,199) in both high and medium resolution.

See the COLOR and PALETTE statements for more information.

If *color* is greater than 3, an **Illegal function call** error is returned.

Examples:

```
10 CLS
20 SCREEN 1
30 FOR I = 0 TO 100
40 PSET (I,I)
50 NEXT
60 LOCATE 14,1
```

Draws a diagonal line from (0,0) to (100,100).

```
40 FOR I = 100 TO 0 STEP -1
50 PSET(I,I),0
60 NEXT I
```

Clears out the line by setting each point to 0.

PRINT Statement

Purpose:

Prints numeric or string *data* on the display.

Syntax:

PRINT *data*[*data*,...]

Comments:

You can substitute a question mark for the word PRINT when using the GW-BASIC program editor.

data is any numeric or string constant or variable. If you omit *data*, GW-BASIC prints a blank line. If you specify more than one data item in the statement, separate the items with commas, semicolons, or spaces. String constants must be enclosed in quotation marks.

For more information about strings, see the STRING\$ function.

Print Positions: GW-BASIC divides each line into 14 print zones of 14 positions each. The position of each data item is determined by the punctuation preceding it:

Separator	Print Position
;	Immediately after last data item
space(s)	Immediately after data item
,	Beginning of next tab zone

If you place a comma, a semicolon, or an SPC or TAB function following the last data item, the next PRINT statement begins printing on the same line, accordingly spaced. If you do not include such trailing punctuation or an SPC or TAB function, GW-BASIC places the cursor at the beginning of the next line.

If GW-BASIC tries to print a string longer than can fit on the current line, it moves to the next line and prints the string. If you print exactly 40 or 80 characters (depending on the screen width set by the WIDTH statement), two lines are skipped unless the PRINT statement ends in with a semicolon.

GW-BASIC prints all numbers with a trailing blank and all positive numbers with a leading blank. Negative numbers are preceded by a minus (-) sign. Single-precision numbers are represented with seven or fewer digits in a fixed-point or integer format.

See the **LPRINT** and **LPRINT USING** statements for information on sending data to be printed on a printer.

Examples:

```
10 X$ = STRING$(10,45)
20 PRINT X$"MONTHLY REPORT" X$
```

Displays:

```
-----MONTHLY REPORT-----
```

The value 45 is the decimal equivalent of the ASCII symbol for the minus sign (-).

PRINT USING Statement

Purpose:

To print *data* on the screen, using a specified *format*.

Syntax:

```
PRINT USING format;data[data,...]
```

Comments:

format consists of one or more field specifier(s) or any alphanumeric character. *format* must be enclosed in quotation marks.

data can be string and/or numeric value(s). If you specify more than one data item in the statement, use the same separators described in PRINT.

Specifiers for String Fields

! prints only the first character in the string.

\spaces\ prints 2 + *n* characters, where *n* is the number of spaces between the slashes. For example, if you type the backslashes without any spaces, GW-BASIC prints two characters; with one space, GW-BASIC prints three characters. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.

```
10 A$="LOOK":B$="OUT"  
30 PRINT USING "!";A$;B$  
40 PRINT USING"\ \";A$;B$  
50 PRINT USING"\ \";A$;B$;"!!"
```

Displays:

```
LOOKOUT  
LOOK OUT!!
```

& Specifies a variable length string field. When the field is specified with &, the string is output exactly as input.

```
10 A$="LOOK":B$="OUT"  
20 PRINT USING "!";A$  
30 PRINT USING "&";B$
```

Displays:

LOUT

Specifiers for Numeric Fields: You can use the following special characters to format the numeric field:

Use number signs to indicate the number of digit positions you want to be printed. GW-BASIC prints one digit for each number sign. It rounds numbers as necessary.

GW-BASIC fills as many digit positions as you indicate. If the number to be printed has fewer digits than specified in the format, the number is right-justified (preceded by spaces) in the field.

You can insert a decimal point at any position in the field. If the format string specifies that a digit is to precede the decimal point, GW-BASIC always prints that digit (0, if necessary).

```
PRINT USING "##.##";.78
```

GW-BASIC prints: 0.78

```
PRINT USING "###.##";987.654
```

GW-BASIC prints: 987.65

```
PRINT USING "##.##" ;10.2,5.3,66.789,.234
```

GW-BASIC prints: 10.20 5.30 66.79 0.23

In the last example, three spaces are inserted at the end of the format string to separate the printed values on the line.

- +** A plus sign typed at the beginning or end of the format string causes GW-BASIC to print the sign (plus or minus) of the number.

`PRINT USING "+##.##";-68.95,2.4,55.6,-9`

GW-BASIC prints: -68.95 + 2.40 + 55.60 -0.90

- A plus sign typed at the end of the format string causes GW-BASIC to print a negative sign following negative numbers.

`PRINT USING"##.##-";-68.95,22.449,-7.01`

GW-BASIC prints: 68.95 22.45 7.01-

- **** Two asterisks typed at the beginning of the format string cause GW-BASIC to fill leading spaces in the numeric field with asterisks. The two asterisks count as digit positions.

`PRINT USING "***#.##";12.39,-0.9,765.1`

GW-BASIC prints: *12.4* -09765.1

- \$\$** A double dollar sign at the beginning of the format string causes BASIC to print a dollar sign to the immediate left of the number. The \$\$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with \$\$. Negative numbers can be used only if the minus sign trails to the right.

`PRINT USING "$$###.##";456.78`

GW-BASIC prints: \$456.78

- ***\$** The ***\$ at the beginning of a format string combines the effects of ** and \$\$. Leading spaces are filled with asterisks, and a dollar sign is printed before the number. ***\$ specifies three more digit positions, one of which is the dollar sign.

`PRINT USING "***$###.##";2.34`

GW-BASIC prints: ***\$2.34

A comma to the left of the decimal point in a formatting string causes GW-BASIC to print a comma to the left of every third digit left of the decimal point. A comma at the end of the format string prints as part of the string.

```
PRINT USING "####.##";1234.5
```

GW-BASIC prints: 1234.50

Four carets placed after the digit position characters specify exponential format. The four carets allow space for GW-BASIC to print E + *nn* (indicating exponential format). You can specify any decimal point position. The significant digits are left-justified, and the exponent is adjusted. Unless you specify a leading plus sign (or a trailing plus sign or minus sign), GW-BASIC uses one digit position to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.## ^ ^ ^ ^";234.56
```

GW-BASIC prints: 2.35E + 02

```
PRINT USING ".#### ^ ^ ^ ^-";888888
```

GW-BASIC prints: .8889E + 06

```
PRINT USING "+.## ^ ^ ^ ^";123
```

GW-BASIC prints: +.12E + 03

Note that the comma is **not** used as a delimiter with the exponential format.

An underscore in the format string causes GW-BASIC to output the next character as a literal character. If you want a literal underscore, use “_” in the string.

```
PRINT USING "_!##.##_!";12.34
```

GW-BASIC prints: !12.34!

If the number to be printed is larger than the specified numeric field, GW-BASIC prints a percent sign (%) in front of the number. If rounding causes the number to exceed the field, a percent sign is printed in front of the rounded number. For example:

`PRINT USING "###.##";111.22`

GW-BASIC prints: %111.22

`PRINT USING ".##";.999`

GW-BASIC prints: %1.00

If the number of digits specified exceeds 24, an *Illegal function call error* results.

PRINT# and PRINT# USING Statements

Purpose:

To write data to a sequential disk file.

Syntax:

```
PRINT#file number,[USING format;] data[, data,...]
```

Comments:

file number is the number used when the file was opened for output.

format consists of the formatting characters described in the PRINT USING statement.

data consists of the numeric and/or string expressions to be written to the file.

Double quotation marks are used as delimiters for numeric and/or string expressions. The first double quotation mark opens the line for input. The second double quotation mark closes it.

To print data as input, enclose it in quotation marks. If you omit the quotation marks, GW-BASIC prints the value assigned to the expression. If no value is assigned, GW-BASIC assumes 0. The quotation marks do not appear on the screen. For example:

```
10 PRINT#1,A
```

Prints 0.

```
10 A = 26
```

```
20 PRINT#1,A
```

Prints 6.

```
10 A = 26
```

```
20 PRINT#1,"A"
```

Prints A.

If double quotation marks are required in a string, use CHR\$(34) (the ASCII character for quotation marks). For example:

```
100 PRINT#1,"He said,"Hello", I think"
```

Prints He said, 0, I think because GW-BASIC assigns the value 0 the variable "Hello."

```
100 PRINT#1, "He said, "CHR$(34)
"Hello,"CHR$(34) " I think."
```

Prints He said, "Hello," I think.

If a string contains commas, semicolons, or significant leading blanks, enclose the string in quotation marks. The following example inputs "CAMERA" to A\$ and "AUTOMATIC 93604-1" to B\$:

```
10 A$="CAMERA,AUTOMATIC":B$="93604-1"
20 PRINT#1,A$,B
30 INPUT#1,A$,B$
```

To separate these strings properly, write double quotation marks, using CHR\$(34). For example:

```
40 PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$; CHR$(34)
"CAMERA,AUTOMATIC""93604-1"
```

The PRINT# statement can be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$$$.##.";J;K;L
```

PRINT# does not compress data on the diskette. It writes it to the diskette exactly as it would write it to the screen. For this reason, be sure to delimit the data on the diskette so that it is input correctly from the diskette.

Use semicolons as shown to separate numeric data items:

```
PRINT#1,A;B;C;X;Y;Z
```

If you use commas as delimiters between strings, the extra blanks inserted between print fields are also written to the diskette. Commas have no effect, however, if used with the exponential format.

Use explicit delimiters, such as semicolons, to separate string data items so that they are formatted correctly on the diskette. For example, avoid:

```
10 A$="CAMERA":B$="93604-1"  
20 PRINT#1,A$,B$
```

The preceding program lines do not input the strings separately, so they yield the diskette image of:

```
CAMERA93604-1
```

To correct the problem, use:

```
20 PRINT#1,A$," ";B$
```

The new line gives a diskette image of:

```
CAMERA,93604-1
```

Which can be read back into two string variables.

PRESET Statement

Purpose:

To clear a point at a specified location in graphics mode. To set a point, see PSET.

Syntax:

PRESET [STEP] (*x*, *y*)[*color*]

Comments:

x is the horizontal coordinate of the point. *y* is the vertical coordinate.

color is the color of the point.

Specify absolute coordinates, or use the STEP option to specify relative coordinates. STEP indicates that *x* and *y* are offsets relative to the last point referenced. For example:

STEP(10,10)

Coordinate values can be beyond the edge of the screen. However, values outside the integer range -32768 to 32767 cause an **Overflow** error.

The upper left corner is always (0,0). The lower left corner is (0,199) in both high and medium resolution.

See the **COLOR** and **PALETTE** statements for more information.

If *color* is greater than 3, an **Illegal function call** error is returned.

PUT Statement (Files)

Purpose:

To write a record from a random buffer to a random disk file.

Syntax:

PUT[#]*file number*[*record number*]

Comments:

file number is the number used to open the file.

record number specifies the record to write. If you omit it, GW-BASIC uses the next available record number (after the last PUT). The largest possible *record number* is $2^{32}-1$. This allows large files with short records. The smallest possible *record number* is 1.

You can use the PRINT#, PRINT# USING, LSET, RSET, or WRITE# statements to put characters in the random file buffer before using a PUT statement.

In the case of WRITE#, GW-BASIC pads the buffer with spaces up to a carriage return.

Any attempt to read or write past the end of the buffer causes a Field overflow error.

PUT can be used for communications files. Here, *record number* is the number of bytes written to the file. *record number* must be less than or equal to length of the buffer set in OPEN "COM(*n*).

PUT Statement (Graphics)

Purpose:

To transfer an image stored in an array to the screen.

Syntax:

PUT(*x*, *y*), *array*[, *action*]

Comments:

The GET and PUT graphics statements make possible animation and high-speed object motion in graphics modes. GET transfers the screen image described by specified points of the rectangle into the array. PUT transfers the image stored in the array onto the screen.

x and *y* are the coordinates of the upper left corner of the image to be transferred. An illegal function call error results if the image to be transferred is too large to fit onto the screen.

action sets the type of interaction between the transferred image and the image already on the screen. It can be PSET, PRESET, AND, OR, or XOR. If you omit *action*, GW-BASIC assumes XOR.

- PSET transfers the data to the screen exactly as it was stored in the array.
- PRESET produces an inverse image (black on white) on the screen.
- AND transfers the image over an existing image. The result is a logical AND of the array and the image on the screen. If no image exists on the screen, AND does not transfer the array image.
- OR superimposes the image on the existing image. The result is a logical OR of the array and the image on the screen.
- XOR is especially useful 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 you put an image against a complex background twice, the background is restored unchanged. Thus, you can move an object without obliterating the background.

For more information about effects within the different modes, see the COLOR, PALETTE, and SCREEN statements.

Animation of an object is usually performed as follows:

1. Put the object(s) on the screen.
2. Recalculate the new position of the object(s).
3. Put the object(s) on the screen a second time at the old location(s) to remove the old image(s).
4. Repeat Step 1, putting the object(s) at the new location(s).

Animation performed in the preceeding manner leaves the background unchanged. You can cut down flicker by minimizing the time between Steps 4 and 1 and by ensuring enough time between Steps 1 and 3. If you are animating more than one object, process every object at the same time, one step at a time.

If preserving the background is not important, you can use PSET instead of XOR as the *action*.

When you first get the image, leave a border around it as large as or larger than the maximum distance the object will move. When you move the object, the border effectively erases any points. This method might be faster than the XOR method described above because only one PUT is required to move an object. However, the image to be put must be larger than the existing image.

Examples:

```
10 CLS:SCREEN 1
20 PSET (130,120)
30 DRAW "U25;E7;R20;D32;L6;U12;L14"
40 DRAW "D12;L6":PSET(137,102)
50 DRAW "U4;E4;R8;D8;L12"
60 PSET (137,88)
70 DRAW "E4;R20;D32;G4":PAINT (131,119)
80 DIM A (500)
90 GET (125,130)-(170,80),A
100 FOR I = 1 TO 1000:NEXT I
110 PUT (20,20),A,PSET
120 FOR I = 1 TO 1000:NEXT I
130 GET (125,130)-(170,80),A
140 FOR I = 1 TO 1000:NEXT I
150 PUT (220,130),A,PRESET
```


RANDOMIZE Statement

Purpose:

To reseed the random number generator.

Syntax:

```
RANDOMIZE [value]  
RANDOMIZE TIMER
```

Comments:

value can be a numeric value in the range -32768 to 32767, or an expression or numeric formula returning a value in that range. If you omit *value*, GW-BASIC suspends program execution and asks for a value by displaying:

Random number seed (32768 to 32767)?

RANDOMIZE [*value*] does not force floating-point values to integer.

If you do not reseed the random number generator, RND returns the same sequence of random numbers each time the program runs.

To change the sequence of random numbers every time the program runs, place a RANDOMIZE statement at the beginning of the program, and change *value* with each run. (See the RND function.)

To get a new random seed without prompting, use the numeric TIMER function as follows:

```
RANDOMIZE TIMER
```

Examples:

The internal clock can be set at intervals:

```
10 RANDOMIZE TIMER
20 FOR I = 1 to 5
30 PRINT RND;
40 NEXT I

RUN
.88598 .484668 .586328 .119426 .709225
Ok

RUN
.803506 .162462 .929364 .292443 .322921
Ok
```

The internal clock can be used for random number seed:

```
5 N = VAL(MID$(TIME$,7,2)) 'get seconds for seed
10 RANDOMIZE N             'install number
20 PRINT N                  'print seconds
30 PRINT RND                 'print random number generated

RUN
36
.2466638
OK

RUN
37
.6530511
OK

RUN
38
5.943847E + 02
OK

RUN
40
.8722131
OK
```

READ Statement

Purpose:

To read values from a DATA statement and assign them to variables.

Syntax:

```
READ variable[,variable,...]
```

Comments:

READ statements assign values from the DATA statement on a one-to-one basis. The first time the program executes READ, it assigns the first variable the first value in the first DATA statement; the second time, it assigns the second value to the second variable; and so on.

READ statement variables can be numeric or string. They must agree in type with the values read. If they do not, GW-BASIC returns a Syntax error.

A single READ statement can access one or more DATA statements, or several READ statements can access one DATA statement. If a program contains multiple DATA statements, GW-BASIC reads the statements in the order in which they appear in the program.

If the number of variables in the list is greater than the number of elements in the DATA statement(s), GW-BASIC returns an Out of data error. If the number of variables is less than the number of elements, subsequent READ statements begin reading data at the first unread element. If there are no subsequent READ statements, GW-BASIC ignores the extra data.

To reread DATA statements from the start, use the RESTORE statement.

Examples:

The following program segment reads the values from the DATA statements into Array A. After execution, the value of A(1) is 3.08, the value of A(2) is 5.19, the value of A(3) is 3.12, and so on. The DATA statement (Lines 110-120) can be placed anywhere in the program.

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

The following program reads string and numeric data from the DATA statement in Line 30.:

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

The program displays:

```
CITY STATE ZIP  
DENVER,COLORADO 80211
```

REM Statement

Purpose:

To allow you to insert remarks (comments) in a program.

Syntax:

```
REM[comment]  
'[comment]
```

Comments:

The keyword REM instructs the computer to ignore everything up to the next line number or the end of the program. This lets you insert comments in your program. When you list the program, GW-BASIC lists the REM statement, including the comment, without executing the REM statement.

You can substitute an apostrophe (') as an abbreviation for REM.

You can use a GOTO or GOSUB to branch to a REM statement, and execution continues with the first executable statement following the REM. However, execution is faster if you skip the REM statement and branch to the next executable statement instead.

To add a remark to the end of a program line, precede the remark with an apostrophe. (Do not use REM in this case.)

Note: Do not use REM in a DATA statement. The program interprets it as legal data.

Examples:

```
.  
.   
.   
120 REM CALCULATE AVERAGE VELOCITY  
130 FOR I = 1 TO 20  
440 SUM = SUM + V(I)  
450 NEXT I
```

or

```
.  
.   
.   
129 FOR I = 1 TO 20 'CALCULATED AVERAGE VELOCITY  
130 SUM = SUM + V(I)  
140 NEXT I
```

RENUM Command

Purpose:

To renumber the program currently in memory.

Syntax:

```
RENUM[new line],[old line],[increment]]
```

Comments:

new line is the first line number for the new sequence. The default is 10.

old line is the program line renumbering begins. The default is the first line.

increment is the increment for the new sequence. The default is 10.

RENUM also changes all line number references following ELSE, GOTO, GOSUB, THEN, ON/GOTO, ON/GOSUB, RESTORE, RESUME, and ERL as needed. If a nonexistent line number appears after one of these statements, GW-BASIC returns Undefined line *x* in *y*, where *x* is the undefined line number and *y* is the number of the line in which the reference appears. RENUM does not change *x* but might change *y*.

You cannot use RENUM to reorder program lines. For example, if a program has Lines 10, 20, and 30, RENUM 15,30 is illegal because it would place Line 30 before Line 20. Also, RENUM cannot create line numbers greater than 65529. If you attempt to do this, GW-BASIC returns an Illegal function call error and leaves the program unchanged.

Examples:

RENUM

Renumbers the entire program, using an increment of 10. The first new line number is 10.

RENUM 300,,50

Renumbers the entire program, using an increment of 50. The first new line number is 300.

RENUM 1000,900,20

Renumbers from Line 900 through the end of the program, using an increment of 20. The first new line number is 1000.

RESET Command

Purpose:

To close all disk files and write the directory information to a disk.

Syntax:

RESET

Comments:

RESET closes all open files on all disks and writes the directory track to every disk with open files.

RESTORE Statement

Purpose:

To allow DATA statements to be reread from a specified *line*.

Syntax:

RESTORE[*line*]

Comments:

If you specify *line*, the next READ statement accesses the first item in the specified DATA statement.

If you omit *line*, the next READ statement accesses the first item in the first DATA statement.

Examples:

```
10 READ A,B,C,  
20 RESTORE  
30 READ D,E,F  
40 DATA 57,68,79
```

```
.  
.   
.
```

Assigns the value 57 to Variable A and D, 68 to Variable B and E, and so on.

RESUME Statement

Purpose:

To continue program execution after an error-recovery procedure.

Syntax:

```
RESUME  
RESUME 0  
RESUME NEXT  
RESUME line
```

Comments:

Select the syntax according to where you want execution to resume:

- RESUME or RESUME 0—to resume at the statement that caused the error
- RESUME NEXT—to resume at the statement immediately following the one that caused an error
- RESUME *line*—to resume at the specified *line*

A RESUME statement that is not in an error-trapping routine causes GW-BASIC to display a RESUME without error message.

Examples:

```
10 ON ERROR GOTO 900  
.  
.  
.  
900 IF (ERR = 230)AND(ERL = 90) THEN PRINT "TRY  
    AGAIN":RESUME 80  
.  
.  
.
```

If an error occurs after execution of Line 10, the program performs the error-recovery procedure in Line 900 and then continues execution at Line 80.

RETURN Statement

Purpose:

To return from a subroutine.

Syntax:

RETURN [*line*]

Comments:

RETURN without *line* causes GW-BASIC to branch from the subroutine back to the statement following the most recent GOSUB statement.

A subroutine can contain more than one **RETURN** statement to return from different points in the subroutine.

RETURN *line* is primarily useful with event trapping. It sends the event-trapping routine back to *line* while still eliminating the GOSUB entry that the trap created.

When you trap a particular event, the trap automatically causes a **STOP** on that event so that recursive traps never take place. The return from the trap routine automatically performs an **ON** unless you specify **OFF** inside the trap routine.

Use caution when specifying *line* with **RETURN**. Any **GOSUB**, **WHILE**, or **FOR** statement active at the time of the trap remains active.

RIGHT\$ Function

Purpose:

To return the specified *number* of characters from the far right portion of *string*.

Syntax:

RIGHT\$(*string*, *number*)

Comments:

number is an integer in the range 1-255.

If *number* is equal to or greater than the length of *string*, RIGHT\$ returns *string*. If *number* equals zero, GW-BASIC returns the null string (length zero). (See the MID\$ and LEFT\$ functions.)

Examples:

```
10 A$="DISK BASIC"  
20 PRINT RIGHT$(A$,5)
```

Displays BASIC, the rightmost five characters in A\$.

RMDIR Command

Purpose:

To delete the subdirectory specified by *pathname*.

Syntax:

RMDIR *pathname*

Comments:

pathname is a string expression, not exceeding 63 characters, identifying the subdirectory to be removed from its parent.

The subdirectory to be deleted must be empty of all files except "." and "..". Otherwise, GW-BASIC returns a Path file/access error.

Examples:

Referring to the sample directory structure illustrated in CHDIR, the following command deletes the subdirectory Report:

RMDIR "SALES\JOHN\REPORT"

RND Function

Purpose:

To return a random number in the range 0-1.

Syntax:

RND[(*number*)]

Comments:

GW-BASIC uses the current seed when generating a random number and produces the same sequence of random numbers each time the program runs unless you reseed the random number generator. (See the RANDOMIZE statement.)

If *number* is 0, RND repeats the last number.

If you omit *number*, or if *number* is greater than 0, RND returns the next random number in the sequence.

If *number* is negative, RND starts the sequence of random numbers at the beginning.

To get a random number in the range 0-*n*, use the following formula:

$\text{INT}(\text{RND} * (n + 1))$

The random number generator can be seeded by using a negative value for *number*.

Examples:

```
10 FOR I = 1 TO 5
20 PRINT INT(RND*101);
30 NEXT
```

Generates five pseudo random numbers in the range 0-100:

53 30 31 51 5

RSET Statement

Purpose:

To move *data* from memory to a random file buffer and justify it to the right in preparation for a PUT statement.

Syntax:

RSET *field name* = *data*

Comments:

field name is a string variable defined in a FIELD statement.

This statement is similar to LSET. The only difference is that RSET right-justifies data in the buffer.

Before using RSET, you must use FIELD to set up buffer fields.

See LSET for details.

RUN Command

Purpose:

To execute the program currently in memory or to load a file from the diskette into memory and run it.

Syntax:

```
RUN [line][,r]  
RUN "filename"[,r]
```

Comments:

RUN or RUN [*line*] runs the program currently in memory. If you specify *line*, execution begins on that line. Otherwise, execution begins at the lowest line number. If there is no program in memory, GW-BASIC returns to the command level.

RUN "*filename*" runs the specified disk file. GW-BASIC closes all open files and deletes the current contents of memory before loading the specified file into memory and executing it.

The *r* option keeps all data files open.

Executing the RUN command turns off any sound that is currently running and resets PLAY to music foreground. It also resets the PEN and STRIG statements to OFF.

Examples:

```
RUN "NEWFIL",R
```

Runs Newfil without closing data files.

SAVE Command

Purpose:

To save the specified file on disk.

Syntax:

```
SAVE "filename"[,a]  
SAVE "filename"[,p]
```

Comments:

filename is a standard filename, enclosed in quotation marks. If *filename* already exists, GW-BASIC writes over the existing file. If you omit *filename*, GW-BASIC uses the extension .bas.

The a option saves the file in ASCII format. If you omit this option, GW-BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some GW-BASIC disk access commands (such as MERGE) and some MS-DOS commands (such as TYPE) require an ASCII format file.

The p option protects the file by saving it in an encoded binary format. When you later run or load a protected file, any attempt to list or edit it fails. If you use the p option, first make an additional copy under another name or on another disk to let you make future program changes.

Examples:

```
SAVE "COM2",A
```

Saves the file Com2.bas in the ASCII format.

```
SAVE "PROG",P
```

Saves the file Prog.bas in binary format, and protects access.

SCREEN Function

Purpose:

To return the ASCII code (0-255) for the character at the specified *row* (line) and *column* on the screen.

Syntax:

$x = \text{SCREEN}(\text{row}, \text{column}[, z])$

Comments:

x is a numeric variable to receive the ASCII code returned.

row is a valid numeric expression in the range 1-25.

column is a valid numeric expression in the range 1-40 or 1-80, depending upon screen width setting. See the WIDTH statement.

z is a valid numeric expression with a true (non-zero) or false (zero) value. It can be used only in text mode.

SCREEN stores the ordinal of the character at the specified coordinates in the numeric variable. In text mode, if you specify the optional parameter *z* and it is true (nonzero), GW-BASIC returns the color attribute for the character. (See the COLOR statement.)

If you enter an out-of-range value, GW-BASIC returns an **Illegal function call** error. You can refer to Row 25 only if you have used KEY OFF to turn off the function key display.

Examples:

100 X = SCREEN (10,10)

Returns the ASCII code of the character at 10,10 in X. If the character is A, X is 65.

110 X = SCREEN (1,1,1)

Returns the color attribute of the character in the upper left corner of the screen.

SCREEN Statement

Purpose:

To set the specifications for the display screen.

Syntax:

```
SCREEN [mode] [, [colorswitch]] [, [apage]] [, [vpape]]
```

Comments:

SCREEN allows you to select a screen mode appropriate for a particular display-hardware configuration. Supported hardware configurations and screen modes are described below.

Monochrome Monitor– Screen Mode 0: You can connect a monochrome display adapter only to a monochrome display. Programs written for this configuration must be in text mode (Screen Mode 0).

CGA with Color Monitor– Screen Modes 0, 1, and 2: Typically, you pair the Color Graphics Adapter (CGA) with a color monitor. This hardware configuration permits running of text mode (Mode 0) and both medium-resolution (Mode 1) and high-resolution (Mode 2) graphics programs.

EGA with Color Monitor– Screen Modes 0, 1, 2, 7, and 8: Screen Modes 0, 1, 2, 7, and 8 let you to use a color monitor connected to an Enhanced Graphics Adapter (EGA). If EGA switches are set for CGA compatibility, programs written for Modes 1 and 2 run just as with the CGA. Modes 7 and 8 are similar to Modes 1 and 2, except that they permit a wider range of colors.

EGA with Enhanced Color Monitor– Modes 0, 1, 2, 7, and 8: With the EGA enhanced display configuration, Modes 0, 1, 2, 7, and 8 are virtually identical to their EGA/color monitor counterparts. Two possible differences are:

- In Screen Mode 0, the border color cannot be the same as for the EGA/color monitor because the border cannot be set on an Enhanced Color Display when it is in 640 x 350 text mode.
- The quality of the text is better on the Enhanced Color Display (an 8 x 14 character box for an Enhanced Color Display vs. an 8 x 8 character box for a color monitor).

EGA with Enhanced Display– Screen Mode 9: Screen Mode 9 takes full advantage of all the capabilities of the Enhanced Color Display. It allows the highest resolution possible for the EGA/Enhanced Color Display configuration. Programs written for Mode 9 do not work for any other hardware configuration.

EGA with Monochrome Display– Screen Mode 10: In Screen Mode 10, you can use a monochrome monitor to display very high resolution monochrome graphics. Programs written for Mode 10 do not work for any other hardware configuration.

Arguments: *mode* is an integer expression that specifies the screen mode: 0, 1, 2, 7, 8, 9, or 10. All other values are illegal. The specific *mode* depends primarily on the hardware that you intend the program to run on, as described above.

The various screen modes are described in detail in the following information.

Screen Mode 0

- Text mode only
- Either 40 x 25 or 80 x 25 text format with character-box size of 8 x 8 or 8 x 14 with EGA
- Assignment of 16 colors to either of two attributes
- Assignment of 16 colors to any of 16 attributes (with EGA)

Screen Mode 1

- 320 x 200 pixel medium-resolution graphics
- 80 x 25 text format with character-box size of 8 x 8
- Assignment of 16 colors to any of four attributes
- Supports both EGA and CGA
- Two bits per pixel

Screen Mode 2

- 640 x 200 pixel high-resolution graphics
- 40 x 25 text format with character-box size of 8 x 8
- Assignment of 16 colors to either of two attributes
- Supports both EGA and CGA
- One bit per pixel

Screen Mode 7

- 320 x 200 pixel medium-resolution graphics
- 40 x 25 text format with character-box size of 8 x 8
- Two, four, or eight memory pages with 64K, 128K, or 256K bytes of memory, respectively, installed on the EGA
- Assignment of any of 16 colors to 16 attributes
- EGA required
- Four bits per pixel

Screen Mode 8

- 640 x 200 pixel high-resolution graphics
- 80 x 25 text format with character-box size of 8 x 8
- One, two, or four memory pages with 64K, 128K, or 256K bytes of memory, respectively, installed on the EGA
- Assignment of any of 16 colors to 16 attributes
- EGA required
- Four bits per pixel

Screen Mode 9

- 640 x 350 pixel enhanced-resolution graphics
- 80 x 25 text format with character-box size of 8 x 14
- Assignment of either 64 colors to 16 attributes (more than 64K of EGA memory) or 16 colors to four attributes (64K of EGA memory)
- Two display pages if 256K of EGA memory installed
- EGA required
- Two bits per pixel (64K EGA memory) or four bits per pixel (more than 64K EGA memory)

Screen Mode 10

- 640 x 350 enhanced-resolution graphics
- 80 x 25 text format with character-box size of 8 x 14
- Two display pages if 256K of EGA memory installed
- Assignment of up to nine pseudo-colors to four attributes; refer to the following tables
- EGA required
- Two bits per pixel

The default attributes for Screen Mode 10 are:

Attribute Value	Displayed Pseudo-Color
0	Off
1	On, normal intensity
2	Blink
3	On, high intensity

The default colors for Screen Mode 10 are:

Color Value	Displayed Pseudo-Color
0	Off
1	Blink, off to on
2	Blink, off to high intensity
3	Blink, on to off
4	On
5	Blink, on to high intensity
6	Blink, high intensity to off
7	Blink, high intensity to on
8	High intensity

For both composite monitors and TVs, *colorswitch* is a numeric expression that is either true (non-zero) or false (zero). A value of zero disables color and permits display of black and white images only. A non-zero value permits color. The meaning of *colorswitch* is inverted in Screen Mode 0.

For hardware configurations that include an EGA and enough memory to support multiple-screen pages, the *apage* and *vpage* options are available. These options determine the *active* and *visual* memory pages. The active page is the area in memory in which graphics statements are written. The visual page is the area of memory that is displayed on the screen.

You can perform animation by alternating the display of graphics pages. The goal is to display the visual page with completed graphics output, while executing graphics statements in one or more active pages.

GW-BASIC displays a page only when graphics output to that page is complete. Thus, the following program fragment is typical:

```
SCREEN 7,,1,2      'work in Page 1, show Page 2
.
.                  'Graphics output to Page 1
.                  'while viewing Page 2
.
SCREEN 7,,2,1      'work in Page 2, show Page 1
.
.                  'Graphics output to Page 2
.                  'while viewing Page 1
.
```

The number of pages available depends on the screen mode and the amount of available memory:

Mode	Resolution	Attribute Range	Color Range	EGA Memory	Pages	Page Size
0	40-column text	NA	0-15 ^b	NA	1	2K
	80-column text	NA	0-15 ^b	NA	1	4K
1	320 x 200	0-3 ^a	0-3	NA	1	16K
2	640 x 200	0-1 ^a	0-1	NA	1	16K
7	320 x 200	0-15	0-15	64K	2	32K
				128K	4	
				256K	8	
				64K	1	
8	640 x 200	0-15	0-15	128K	2	64K
				256K	4	
				64K	1	
9	640 x 350	0-3	0-15	64K	1	64K
		0-15	0-63	128K	1	128K
		0-15	0-63	256K	2	
10	640 x 350	0-3	0-8	128K	1	128K
				256K	2	

a Attributes applicable only with EGA.
b Numbers in the range 16-31 are blinking versions of Colors 0-15.

Screen Mode Specifications

Attributes and Colors: For various screen modes and display hardware configurations, different attribute and color settings exist. (See the PALETTE statement for a discussion of attribute and color numbers.) The following table summarizes the majority of these attribute and color configurations:

Attributes for Mode			Color Display		Monochrome Display	
1,9	2	0,7,8,9 ^a	Number ^b	Color	Number ^b	Color
0	0	0	0	Black	0	Off
		1	1	Blue		(Underlined) ^c
		2	2	Green	1	On ^c
		3	3	Cyan	1	On ^c
		4	4	Red	1	On ^c
		5	5	Magenta	1	On ^c
		6	6	Brown	1	On ^c
		7	7	White	1	On ^c
		8	8	Gray	0	Off
		9	9	Light Blue		High intensity (underlined)
1		10	10	Light Green	2	High intensity
		11	11	Light Cyan	2	High intensity
		12	12	Light Red	2	High intensity
2		13	13	Light Magenta	2	High intensity
		14	14	Yellow	2	High intensity
3	1	15	15	High-intensity White	0	Off

^a With EGA memory greater than 64K

^b Only for Mode 0 (monochrome)

^c Off when used for background

Default Attributes for Screen Modes

The following table lists the default foreground colors for the various modes:

Default Foreground Attribute			Default Foreground Color	
Screen mode	Color/Ext ^a Display	Monochrome Display	Color/Ext ^a Display	Monochrome Display
0	7	7	7	1
1	3	NA	15	NA
2	1	NA	15	NA
7	15	NA	15	NA
8	15	NA	15	NA
9	3 ^b	NA	63	NA
10	NA	3	NA	8
^a = an Enhanced Color Monitor ^b 15 if greater than 64K of EGA memory NA = Not applicable				

Default Foreground Colors

SGN Function

Purpose:

To return the sign of *number*.

Syntax:

SGN(*number*)

Comments:

number is any numeric expression.

If *number* is positive, SGN returns 1.

If *number* is 0, SGN returns 0.

If *number* is negative, SGN returns -1.

Examples:

```
10 INPUT "Enter value",X
20 ON SGN(X) + 2 GOTO 100,200,300
```

GW-BASIC branches to Line 100 if X is negative, Line 200 if X is 0, and Line 300 if X is positive.

SHELL Statement

Purpose:

To load and execute another program or batch file. When the program finishes, control returns to the GW-BASIC program, at the statement following the SHELL statement. The GW-BASIC manual refers to a program executed under control of GW-BASIC as a *child process*.

Syntax:

SHELL [*string*]

Comments:

string is a valid string expression containing the name of the child process and (optionally) command arguments.

The program name in *string* can have any extension that is supported by MS-DOS's Command.com. If you omit the extension, Command.com looks for a .com file, then an .exe file and finally, a .bat file. If it doesn't find such a file, SHELL issues a File not found error.

COMMAND attempts to processes any text that is separated from the program name by at least one blank space as program parameters.

GW-BASIC remains in memory while the child process is running. When the child process finishes, GW-BASIC continues at the statement following the SHELL statement.

SHELL with no *string* returns you to MS-DOS. From there you can now do anything that COMMAND allows. When ready to return to GW-BASIC, type the MS-DOS command EXIT.

Examples:

SHELL

Transfers control to Command.com. You can execute MS-DOS commands such as:

DIR
TIME

and then type **EXIT** to return to GW-BASIC.

Write some data to be sorted, use **SHELL SORT** to sort it, then read the sorted data to write a report.

```
10 OPEN "SORTIN.DAT" FOR OUTPUT AS #1
20 'Write data to be sorted
.
.
.
1000 CLOSE 1
1010 SHELL "SORT < SORTIN.DAT > SORTOUT.DAT"
1020 OPEN "SORTOUT.DAT" FOR INPUT AS #1
1030 'Process the sorted data
```

SIN Function

Purpose:

To calculate the trigonometric sine of *number*, in radians.

Syntax:

`SIN(number)`

Comments:

`SIN(number)` is calculated in single precision unless you use the /d switch when loading GW-BASIC.

You can obtain `SIN(number)` when *number* is in degrees, by using `SIN(number* π /180)`.

Examples:

```
PRINT SIN(1.5)
```

Displays .9974951, the sine of 1.5 radians (single precision).

SOUND Statement

Purpose:

To generate sound through the speaker.

Syntax:

SOUND *frequency, duration*

Comments:

frequency is the desired frequency in hertz (cycles per second). It is a numeric expression in the range 37-32767.

duration is the duration in clock ticks. Clock ticks occur 18.2 times per second. *duration* must be a numeric expression in the range 0-65535.

Duration values below .022 produce an infinite sound until you or the program execute the next SOUND or PLAY statement.

To turn off any active SOUND statement, set *duration* to 0. If GW-BASIC is not running a SOUND statement, a duration of 0 has no effect.

GW-BASIC executes the sound in the foreground or background, depending on PLAY statement.

The following table shows the frequency you specify to generate a note in an octave adjacent to middle C:

Note	Frequency	Note	Frequency
C	130.810	C ^a	523.250
D	146.830	D	587.330
E	164.810	E	659.260
F	174.610	F	698.460
G	196.000	G	783.990
A	220.000	A	880.000
B	246.940	B	987.770
C	261.630	C	1046.500
D	293.660	D	1174.700
E	329.630	E	1318.500
F	349.230	F	1396.900
G	392.000	G	1568.000
A	440.000	A	1760.000
B	493.880	B	1975.500

^aMiddle C

Relationship of Notes and Frequencies

By doubling or halving the frequency, you can estimate the coinciding note values for the preceding or following octave.

To produce periods of silence, use:

SOUND 32767,duration

There are 1092 clock ticks per minute. To calculate the duration of one beat, divide the number beats per minute into 1092..

The following table shows the number of clock ticks for some typical tempos:

Tempo	Notation	Beats per Minute	Ticks per Beat
very slow	Larghissimo		
	Largo	40-66	27.3-18.2
	Larghetto	60-66	18.2-16.55
	Grave		
	Lento		
	Adagio	66-76	16.55-14.37
slow	Adagietto		
	Andante	76-108	14.37-10.11
medium	Andantino		
	Moderato	108-120	10.11-9.1
fast	Allegretto		
	Allegro	120-168	9.1-6.5
	Vivace		
	Veloce		
	Presto	168-208	6.5-5.25
very fast	Prestissimo		

Examples:

```
2500 SOUND RND*1000 + 37,2
2600 GOTO 2500
```

Creates random sounds of short duration.

SPACE\$ Function

Purpose:

To return a string of *number* spaces.

Syntax:

SPACE\$(*number*)

Comments:

number is rounded to an integer and must be in the range 0-255. (See the SPC function.)

Examples:

```
10 FOR N = 1 TO 5
20 X$ = SPACE$(N)
30 PRINT X$;N
40 NEXT N
```

This program displays:

```
1
2
3
4
5
```

Line 20 adds one space for each loop execution.

SPC Function

Purpose:

To skip a specified *number* of spaces in a PRINT or LPRINT statement.

Syntax:

SPC(*number*)

Comments:

number must be in the range 0-255.

If *number* is greater than the defined width of the printer or the screen, GW-BASIC uses *number* modulo *width*. (See "Integer Division and Modulus Arithmetic" in Chapter 6 of the *Tandy GW-BASIC User's Guide*.)

A semicolon is assumed to follow the SPC(*number*) command.

You can use SPC only with PRINT, LPRINT, and PRINT#. (See also the SPACE\$ function.)

Examples:

```
PRINT "OVER" SPC(15) "THERE"
```

Displays:

```
OVER THERE
```

SQR Function

Purpose:

Returns the square root of *number*.

Syntax:

SQR(number)

Comments:

number must be greater than or equal to 0.

GW-BASIC returns the result as a single-precision number unless you specified the /d switch when loading GW-BASIC.

Examples:

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X; SQR(X)
30 NEXT
```

This program displays:

```
10  3.162278
15  3.872984
20  4.472136
25  5
```

STICK Function

Purpose:

To return the horizontal and vertical coordinates of two joysticks.

Syntax:

$x = \text{STICK}(\text{action})$

Comments:

x is a numeric variable for storing the result.

action can be one of the following:

- 0 returns the horizontal (x) coordinate for the left joystick; also stores the x and y values for both joysticks so that you can perform the remaining three *actions*
- 1 returns the vertical (y) coordinate for the left joystick
- 2 returns the horizontal coordinate for the right joystick
- 3 returns the vertical coordinate for the right joystick

STOP Statement

Purpose:

To terminate program execution and return to the command level.

Syntax:

STOP

Comments:

You can use STOP statements anywhere in a program to terminate execution. When GW-BASIC encounters a STOP statement, it displays:

Break in line *nnnnn*

where *nnnnn* is the line number that contains the STOP.

Use the CONT command to continue execution.

Unlike the END statement, STOP does not close files.

Examples:

```
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
```

STR\$ Function

Purpose:

To convert *number* to a string.

Syntax:

STR\$(*number*)

Comments:

STR\$ is the complementary function to VAL.

If *number* is positive, STR\$ places a blank before the string. If *number* is negative, STR\$ places a minus sign (-) before the string.

While arithmetic operations can be performed on *number*, only string functions and operations can be performed on the string.

Examples:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,40,50
```

.
.
.

Branches to various subroutines, depending on the number of characters you type before you press **ENTER**.

STRIG Statement and Function

Purpose:

To return the status of the joystick buttons.

Syntax:

As a statement:

```
STRIG ON  
STRIG OFF
```

As a function:

$x = \text{STRIG}(\text{number})$

Comments:

x is a numeric variable for storing the result.

number is a valid numeric expression in the range 0-7:

- | | |
|---|---|
| 0 | returns -1 if Trigger 1 on the left joystick has been pressed and released since the last STRIG(0) statement; returns 0, if not. |
| 1 | returns -1 if Trigger 1 on the left joystick is currently pressed; returns 0, if not. |
| 2 | returns -1 if Trigger 1 on the right joystick has been pressed and released since the last STRIG(2) statement; returns 0, if not. |
| 3 | returns -1 if Trigger 1 on the right joystick is currently pressed; returns 0, if not. |
| 4 | returns -1 if Trigger 2 on the left joystick has been pressed and released since the last STRIG(4) statement; returns 0 if not. |
| 5 | returns -1 if Trigger 2 on the left joystick is currently pressed; returns 0, if not. |

- 6 returns -1 if Trigger 2 on the right joystick has been pressed and released since the last STRIG(6) statement; returns 0 if not.
- 7 returns -1 if Trigger 2 on the right joystick is currently pressed; returns 0, if not.

STRIG ON must be executed before any STRIG(*number*) function call can be made. Once STRIG ON is executed, GW-BASIC checks before each statement to see whether a button has been pressed.

STRIG(number) Statement

Purpose:

To allow the use of a joystick by enabling or disabling the trapping of its buttons.

Syntax:

STRIG(*number*) ON
STRIG(*number*) OFF
STRIG(*number*) STOP

Comments:

number is 0, 2, 4, or 6, corresponding to the joystick button:

- 0 Trigger 1 on the left joystick
- 2 Trigger 1 on the right joystick
- 4 Trigger 2 on the left joystick
- 6 Trigger 2 on the right joystick

Examples:

STRIG(*number*) ON

Enables trapping of the joystick buttons. After this statement executes, GW-BASIC checks to see whether the specified button has been pressed before executing further statements.

STRIG(*number*) OFF

Disables GW-BASIC from checking the state of the specified button.

STRIG(*number*) STOP

Disables trapping of the specified button with the ON STRIG(*number*) statement. Any pressings are remembered so that trapping can take place once it is re-enabled.

STRING\$ Function

Purpose:

To create a string with a length of *number* characters in which all the characters have ASCII code *character*, or in which all the characters are the same as the first character of *string*

Appendix B in the *Tandy GW-BASIC User's Guide* lists ASCII character codes.

Syntax:

```
STRING$(number, character)  
STRING$(number, string)
```

Comments:

STRING\$ is also useful for printing top and bottom borders on the screen or the printer.

number and *character* are integer expressions in the range 0-255.

Examples:

```
10 X$ = STRING$(10,45)  
20 PRINT X$ "MONTHLY REPORT" X$
```

Uses 45, the decimal equivalent of ASCII symbol for a hyphen, to display:

```
-----MONTHLY REPORT-----  
Ok
```

SWAP Statement

Purpose:

To exchange the values of two variables.

Syntax:

SWAP variable1, variable2

Comments:

You can swap variables of any type (integer, single precision, double precision, or string) as long as the two variables are of the same type. If they are not, GW-BASIC returns a Type mismatch error.

Examples:

```
10 A$="ONE ":B$="ALL ":C$="FOR "  
20 PRINT A$ C$ B$  
30 SWAP A$, B$  
40 PRINT A$ C$ B$
```

Line 30 swaps the values in the A\$ and B\$ strings so that the program displays:

```
ONE FOR ALL  
ALL FOR ONE
```

SYSTEM Command

Purpose:

To return to MS-DOS.

Syntax:

SYSTEM

Comments:

Before you enter SYSTEM, be certain you save the current program. Otherwise, the program is lost.

The SYSTEM command closes all the files before it returns to MS-DOS. If you enter GW-BASIC through a batch file in MS-DOS, the SYSTEM command returns you to the batch file, which continues executing at the point from which it left off.

Examples:

SYSTEM

Returns you to the MS-DOS system prompt.

TAB Function

Purpose:

Spaces to position *number* on the screen.

Syntax:

TAB(*number*)

Comments:

You can use TAB only in PRINT, LPRINT, and PRINT# statements. (See the SPC function.)

number must be in the range 1-255.

If the current print position is already beyond space *number*, TAB goes to that position on the next line.

The leftmost position is 1. The rightmost position is the screen width.

If the TAB function is at the end of a list of data items, GW-BASIC does not return the cursor to the next line. (It acts as if a semicolon follows the TAB function.)

Examples:

```
10 PRINT "NAME" TAB(25) "AMOUNT": PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
```

These program lines display:

```
NAME AMOUNT
G. T. JONES $25.00
```

TAN Function

Purpose:

To calculate the trigonometric tangent of *number*, in radians.

Syntax:

TAN(*number*)

Comments:

GW-BASIC returns the result as a single-precision number unless you use /d when loading GW-BASIC.

If TAN overflows, it displays the Overflow error message, and execution continues.

To obtain the tangent of *number* when *number* is in degrees, use TAN(*number** π /180).

Examples:

10 Y = TAN(X)

Stores the value of the tangent of X radians in Y.

TIME\$ Statement and Variable

Purpose:

To set or retrieve the current time.

Syntax:

As a statement:

`TIME$ = string`

As a variable:

`string = TIME$`

Comments:

string is a valid string literal or variable that lets you set the hours (*hh*), hours and minutes (*hh:mm*), or hours, minutes, and seconds (*hh:mm:ss*).

GW-BASIC uses a 24-hour clock. For example, you need to specify 20:15:00 for 8:15 p.m.

hh sets the hour (0-23). Minutes and seconds default to 00.

hh:mm sets the hour and minutes (0-59). Seconds default to 00.

hh:mm:ss sets the hour, minutes, and seconds (0-59).

If *string* is not a valid string, a Type mismatch error results.

Although you can omit leading zeros in each of the values, you must include at least one digit of the preceding value. For example, you can type 1:5 to set the time to 1:05 a.m. However, :5 is invalid. Similarly, to set the time to a half hour past midnight, use 0:30, not :30.

If any of the values is out of range, GW-BASIC issues an Illegal function call error and retains the previous time.

If TIME\$ is the target of a string assignment, GW-BASIC stores the current time.

If TIME\$ is the expression in a LET or PRINT statement, GW-BASIC retrieves it and assigns it to the string variable .

If *string* = TIME\$, TIME\$ returns an eight-character string in the form *hh:mm:ss*.

Examples:

```
TIME$ = "08:00"
```

Sets the time as 8:00 a.m.

```
PRINT TIME$
```

Displays the current time.

```
10 KEY OFF:SCREEN 0:WIDTH 80:CLS
20 LOCATE 25,5
30 PRINT DATE$,TIME$;
40 SEC = VAL(MID$(TIME$,7,2))
50 IF SEC = SSEC THEN 20 ELSE SSEC = SEC
60 IF SEC = 0 THEN 1010
70 IF SEC = 30 THEN 1020
80 IF SEC < 57 THEN 20
1000 SOUND 1000,2:GOTO 20
1010 SOUND 2000,8:GOTO 20
1020 SOUND 400,4:GOTO 20
```

Displays the current date and time on the 25th line of the screen and generates sound on the minute and half-minute.

TIMER Function

Purpose:

To return the number of seconds elapsed since midnight or the last system reset.

Syntax:

elapsed time = TIMER

Comments:

GW-BASIC always returns the seconds as a single-precision floating-point number.

Fractions of seconds are calculated to the nearest degree possible.

TRON/TROFF Commands

Purpose:

To trace the execution of program statements.

Syntax:

```
TRON
TROFF
```

Comments:

As an aid in debugging, the TRON (trace on) command enables a trace flag that prints the number of each program line as the line executes. It displays the number enclosed in square brackets. TRON can be executed in either the direct or indirect mode.

To disable the trace flag, either use TROFF or execute a NEW command.

Examples:

```
TRON
Ok
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

RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok

TROFF
Ok
```

UNLOCK Statement

Purpose:

To release locks that have been applied to an opened file in a *workgroup*, or network, environment.

Syntax:

UNLOCK [#] *file number* [, [*start record*] [TO *end record*]]

Comments:

file number is the number originally assigned to the file by the program.

You can specify either one record (*start record*) or a range of records (*start record* TO *end record*) to unlock. If you specify a range, *start record* must be less than or equal to *end record*.

The range of legal record numbers is 1 to $2^{32} - 1$. The limit on record size is 32767 bytes.

If you omit *start record*, UNLOCK assumes Record 1.

If you omit *end record*, command unlocks only the specified record.

The following are valid variations of the UNLOCK statement:

UNLOCK #file number

Unlocks the entire file specified by *file number*.

UNLOCK #file number, start record

Unlocks only the specified record.

UNLOCK #file number, TO end record

Unlocks Record 1 through the specified record.

UNLOCK #file number, start record TO end record

Unlocks all records in the range *start record-end record*.

Always unlock the locked file or record range before closing the file. Failure to do so can jeopardize future access to that file in a workgroup environment.

In the case of files opened in RANDOM mode, the range of records specified must match exactly the range given in the LOCK statement.

If GW-BASIC cannot grant a syntactically correct UNLOCK request, the Permission denied message appears. The UNLOCK statement must match exactly the paired LOCK statement.

Because you will probably want to lock files/records only for a short time, we recommend using LOCK within short-term paired LOCK/UNLOCK statements.

Examples:

The following sequence demonstrates how the LOCK/UNLOCK statements should be used:

```
LOCK #1, 1 TO 4  
LOCK #1, 5 TO 8  
UNLOCK #1, 1 TO 4  
UNLOCK #1, 5 TO 8
```

The following example is illegal:

```
LOCK #1, 1 TO 4  
LOCK #1, 5 TO 8  
UNLOCK #1, 1 TO 8
```

USR Function

Purpose:

To call an assembly language subroutine and pass *argument* to that subroutine.

Syntax:

$v = \text{USR}[\text{number}](\text{argument})$

Comments:

Although the CALL statement is recommended for calling assembly language subroutines, you can also use the USR function. See Appendix C in the *Tandy GW-BASIC User's Guide* for a comparison of CALL and USR and for a detailed discussion of calling assembly language subroutines.

number specifies the USR routine being called. It can be any number in the range 0-9. (USR lets you call as many as ten assembly language subroutines and then continue execution of your GW-BASIC program.) If you omit *number*, GW-BASIC assumes 0. (See DEF USR for the rules governing *number*).

argument can be any numeric or string expression.

Before you can execute a USR function, you must define the USR call offset in a corresponding DEF USR statement. This offset and the currently active DEF SEG segment address determine the starting address of the subroutine.

If you require more than ten user routines, you can redefine the value(s) of DEF USR for the other starting addresses as many times as needed.

If you use a segment other than the default segment (BASIC data segment, DS), you must execute a DEF SEG statement prior to a USR call. This ensures 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.

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed. If no argument is required by the assembly language routine, you must supply a dummy argument.

VAL Function

Purpose:

Returns the numerical value of *string*.

Syntax:

VAL(*string*)

Comments:

The VAL function also strips leading blanks, tabs, and line feeds from *string*. For example, the following line returns -3:

```
VAL("-3")
```

VAL is the complement to the STR\$ function (for numeric-to-string conversion).

If the first character of *string* is not numeric, VAL returns 0.

Examples:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$) < 90000 OR VAL(ZIP$) > 96699 THEN
   PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$) > = 90801 AND VAL(ZIP$) < = 90815 THEN
   PRINT NAME$ TAB(25) "LONG BEACH"
.
.
.
```

Searches for zip codes in the specified ranges to determine whether they are in Long Beach or are "out of state."

VARPTR Function

Purpose:

To return the address in memory of *variable* or the file control block (FCB).

Syntax:

VARPTR(*variable*)
VARPTR(*#file number*)

Comments:

Use VARPTR to obtain the address of a variable or array so you can pass it to an assembly language subroutine.

You can pass a function call in the form:

VARPTR(A(0))

To pass an array and return the lowest-addressed element of the array to the routine.

The addresses of the arrays change whenever you assign a new simple variable, so be sure to assign all simple variables before calling VARPTR for an array.

VARPTR (*#file number*) returns the starting address of the GW-BASIC file control block assigned to *file number*.

VARPTR (*variable*) returns the address of the first byte of data identified with *variable*.

You must assign a value to *variable* before executing VARPTR. Otherwise, an illegal function call error results.

You can use any type (numeric, string, or array) variable. The address returned is an integer in the range 32767 to -32768. If VARPTR returns a negative address, add it to 65536 to obtain the actual address.

Add the offsets in the following table to the address returned by VARPTR for a file control block to obtain information about the file:

Offsets to FCB Information

Off-

set:	Length:	Name:	Description:
0	1	Mode	The mode in which the file was opened: 1 Input only 2 Output only 4 Random I/O 16 Append only 32 Internal use 64 Future use 128 Internal use
1	38	FCB	Disk file control block.
39	2	CURLOC	For sequential files, the number of sectors read or written. For random-access files, the number of the last record read or written + 1.
41	1	ORNOFS	Number of bytes in sector when read or written.
42	1	NMLOFS	Number of bytes left in INPUT buffer.
43	3	***	Reserved for future expansion.
46	1	DEVICE	Device number: 0-9 Disk drives A: through J: 255 KYBD: 254 SCRN: 253 LPT1: 252 CAS1: 251 COM1: 250 COM2: 249 LPT2: 248 LPT3:
47	1	WIDTH	Device width.
48	1	POS	Position in buffer for PRINT.
49	1	FLAGS	For internal use during BLOAD/BSAVE. Not used for data files.

Off-set:	Length:	Name:	Description:
50	1	OUTPOS	Output position used during tab expansion.
51	128	BUFFER	Physical data buffer. Used to transfer data between DOS and BASIC. Use this offset to examine data in sequential I/O mode.
179	2	VRECL	Variable length record size; set by <i>length</i> in OPEN statement. The default is 128.
181	2	PHYREC	Current physical record number.
183	2	LOGREC	Current logical record number.
185	1	***	Future use.
186	2	OUTPOS	For disk files only; the output position for PRINT, INPUT, and WRITE.
188	n	FIELD	Actual FIELD data buffer. Size is determined by S: switch. VRECL bytes are transferred between BUFFER and FIELD on I/O operations. Use this offset to examine file data in random I/O mode.

Offsets to FCB Information

Examples:

```
100 X = VARPTR(Y)
```

Stores in the variable X an address that points to the storage space assigned to the variable Y.

```
10 OPEN "DATA.FIL" AS #1
20 FCBADR = VARPTR(#1)
30 DATADR = FCBADR + 188
40 A$ = PEEK(DATADR)
```

In Line 20, FCBADR contains the start of the FCB.

In Line 30, DATADR contains the address of the data buffer.

In Line 40, A\$ contains the first byte in the data buffer.

VARPTR\$ Function

Purpose:

To return a character form of the offset of *variable* in memory.

Syntax:

VARPTR\$(*variable*)

Comments:

variable is the name of a variable that exists in the program.

Note: The addresses of the arrays change whenever you assign a new simple variable, so be sure to assign all simple variables before calling VARPTR for an array.

VARPTR\$ returns a three-byte string:

Byte 0 contains one of the following variable types:

- 2 integer
- 3 string
- 4 single precision
- 8 double precision

Byte 1 contains the least significant byte of the 8086 address format.

Byte 2 contains the most significant byte of the 8086 address format.

Examples:

100 X = USR(VARPTR\$(Y))

The following line uses the PLAY subcommand X, plus the contents of A\$, as the argument for PLAY:

10 PLAY "X" + VARPTR\$(A\$)

VIEW Statement

Purpose:

To create a viewport that redefines the screen parameters. This defined area, a *window*, becomes the only area in which you can draw graphics.

Syntax:

```
VIEW [[SCREEN]][(x1, y1)-(x2, y2) [, [color] [, [border]]]]
```

Comments:

RUN or VIEW with no options defines the entire screen as the viewport.

x1 and *y1* are the upper left coordinates of the viewport.

x2 and *y2* are the lower right coordinates of the viewport.

color is the color with which to fill the viewport.

border is an integer expression that specifies the color for the boundary line around the viewport (assuming that there is enough space for the line). If you omit *border*, no border is drawn.

All coordinates must be within the limitations of the screen. GW-BASIC sorts the *x* and *y* coordinate pairs, placing the smallest values first.

If you omit SCREEN, points are plotted relative to the viewpoint, that is, GW-BASIC adds *x1* and *y1* to *x* and *y* before plotting the point.

It is possible to have a varied number of pairs of *x* and *y*. The only restriction is that *x1* cannot equal *x2* and *y1* cannot equal *y2*.

If you include SCREEN, points are plotted absolutely (relative to Point 0,0). Only points in the current viewpoint are plotted.

When using VIEW, the CLS statement clears only the current viewport. To clear the entire screen, use VIEW to disable the viewpoints; then, use CLS to clear the screen. CLS does not move the cursor to home.

Use **CTRL HOME** to send the cursor home and clear the screen.

Examples:

VIEW (10,10)-(200,100)

Defines a viewport such that the statement PSET(0,0),3 would set a point at the physical screen location 10,10.

VIEW SCREEN (10,10)-(200,100)

Defines a viewport such that the point designated by the statement PSET(0,0),3 would not appear because 0,0 is outside the viewport. PSET(10,10),3 is within the viewport, however.

VIEW PRINT Statement

Purpose:

To set the boundaries of the screen text window.

Syntax:

VIEW PRINT [*top line* TO *bottom line*]

Comments:

Once you execute VIEW PRINT, all statements and functions that normally function within the text viewport (for example, CLS, LOCATE, PRINT, and SCREEN) function within the new text screen parameters.

The screen editor also limits functions such as scroll and cursor movement to the text window.

VIEW PRINT without *top line* TO *bottom line* initializes the whole screen area (Lines 1-24) as the text window. By default, Line 25 is not used.

For more information, see VIEW.

Examples:

VIEW PRINT 1 to 15

Defines the first 15 lines of the display as the text viewport.

WAIT Statement

Purpose:

To suspend program execution until the specified machine input *port* develops a specified bit pattern.

Syntax:

WAIT *port*, *number1* [, *number2*]

Comments:

port represents a valid machine port in the range 0-65535.

number1 and *number2* are integer expressions in the range of 0-255.

GW-BASIC reads the data at *port* and XORs it with *number2*, if given. If you omit *number2*, GW-BASIC XORs the data with zero. It then ANDs the result with *number1*.

If the result is zero, GW-BASIC loops back and reads the data at the port again to check for the specified pattern. If the result is nonzero, execution continues with the next statement.

When executed, the WAIT statement tests Byte *number1* for set bits. If any bit is set, the program continues with the next statement in the program. WAIT does not wait for an entire pattern of bits to appear, but only for one of them to occur.

It is possible to enter an infinite loop with the WAIT statement. To exit the loop, press **CTRL** **BREAK** or reset the system.

Examples:

```
100 WAIT 32,2
```

Suspends machine operation until Port 32 receives 2 (Bit 2 is 1) as input.

WHILE-WEND Statement

Purpose:

To execute a series of statements in a loop as long as a given condition is true.

Syntax:

```
WHILE expression
.
.
.
[loop statements]
.
.
.
WEND
```

Comments:

expression is any numeric or string expression, usually making logical or relational comparisons.

If *expression* is true (non-zero), GW-BASIC executes the *loop statements* until it encounters a WEND statement. GW-BASIC then returns to the WHILE statement and checks *expression*. If *expression* is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

You can nest WHILE and WEND loops to any level. Each WEND matches the most recent WHILE.

An unmatched WHILE statement causes a WHILE without WEND error.
An unmatched WEND statement causes a WEND without WHILE error.

Examples:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS = 1
110 WHILE FLIPS
115 FLIPS = 0
120 FOR N = 1 TO J-1
130 IF A$(N) > A$(N+1) THEN SWAP A$(N), A$(N+1): FLIPS = 1
140 NEXT N
150 WEND
```

The program sorts the elements in A\$. It continues until there are no elements to swap in Line 130. At this time, FLIPS = 0 and the WHILE condition is no longer true. Program control then drops to LINE 150.

WIDTH Statement

Purpose:

To set the printed line width, in number of characters, for the screen and printer.

Syntax:

WIDTH *size*
WIDTH *file number, size*
WIDTH "*device*", *size*

Comments:

size, an integer in the range 0-255, is the new width.

file number is the number of the open file.

device is a valid string expression identifying the device. Valid devices are SCRN:, LPT1:, LPT2:, LPT3:, COM1:, and COM2:.

Changing Screen Width: To change the screen width, use either of the following:

WIDTH *size*
WIDTH "SCRN:", *size*

Only a 40- or 80-character column width is allowed.

See the SCREEN statement for more information.

Changing the screen mode affects the screen width only if you are moving between Screen Mode 2 and Screen Mode 1 or 0.

Note: Changing the screen width clears the screen and sets the border screen color to black.

Changing the Printer Width: The following WIDTH statement is used as a deferred width assignment for the printer. This statement stores the new width value without actually changing the current width setting:

WIDTH "LPT1:", *size*

A statement of the following form recognizes this stored width value:

```
OPEN "LPT1:" FOR OUTPUT AS number
```

and uses it while the file is open:

```
WIDTH file number, size
```

If the file is open to LPT1:, the printer width immediately changes to the new size. This allows you to change the width at will while the file is open. This form of WIDTH has meaning only for LPT1:. After outputting the indicated number of characters from the open file, GW-BASIC inserts the carriage return at the end of the line and wraps the output, if the width is less than the length of the record.

Valid widths for the printer are 1-255.

Specifying WIDTH 255 for the printer (LPT1:) enables line wrapping. This has the effect of infinite width.

Any value entered outside of the indicated ranges results in an **Illegal function call error**. GW-BASIC retains the previous value.

Using the WIDTH statement on a communications file causes a carriage return to be sent after the number of characters specified by *size*. It does not alter either the receive or transmit buffer.

Examples:

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
.
.
.
6020 WIDTH #1,40
```

Line 10 stores a printer width of 75 characters per line.

Line 20 opens file #1 to the printer and sets the width to 75 for subsequent PRINT #1,... statements.

Line 6020 changes the current printer width to 40 characters per line.

WINDOW Statement

Purpose:

To redefine the coordinates of the viewport so that you can draw objects that are not restricted by the physical coordinate range.

Syntax:

WINDOW [[SCREEN](x1, y1)-(x2, y2)]

Comments:

x1 and *y1* are single-precision, floating-point numbers representing the upper left *world coordinates*.

x2 and *y2* are single-precision, floating-point numbers representing the lower right world coordinates.

The world coordinates define the space that graphics statements map into the physical coordinate space. For example, using WINDOW to redefine world coordinates, you can cause an object to extend beyond a viewport or to fill only a small portion of the same viewport.

The VIEW statement defines physical coordinate space. It defaults to the entire screen.

The window is the rectangular region in the world coordinate space. It allows zoom and pan and lets you draw lines, graphics, and objects in space not bounded by the logical limits of the screen. When you use WINDOW, GW-BASIC converts the world coordinates into the appropriate physical coordinates for subsequent display within screen space.

If you omit SCREEN, WINDOW inverts the *y* coordinates on subsequent graphics statements. This makes *x1, y1* the lower left point and *x1, y2* the upper right point, allowing you to view the screen in true Cartesian coordinates.

If you include SCREEN, WINDOW does not invert the *y* coordinates.

x1, y1 is the upper left point, and *x2, y2* is the lower right point.

The WINDOW statement sorts the x and y pairs into ascending order:

WINDOW (50,50)-(10,10)

becomes:

WINDOW (10,10)-(50,50)

WINDOW (-2,2)-(-2,-2)

becomes:

WINDOW (-2,-2)-(2,2)

$x1$ cannot equal $x2$, and $y1$ cannot equal $y2$.

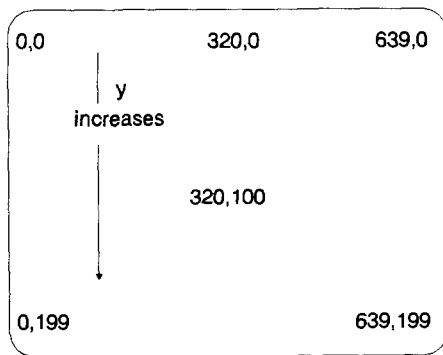
WINDOW with no options disables previous WINDOW statements.

Examples:

If you type:

NEW
SCREEN 2

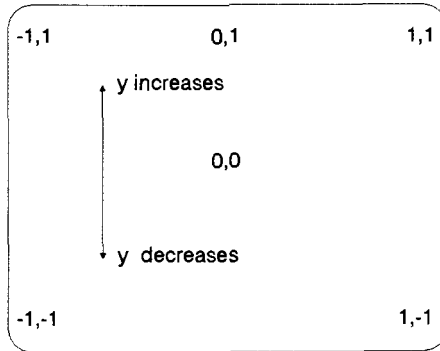
the screen uses the standard coordinate attributes as follows:



If you type:

WINDOW (-1,-1)-(1,1)

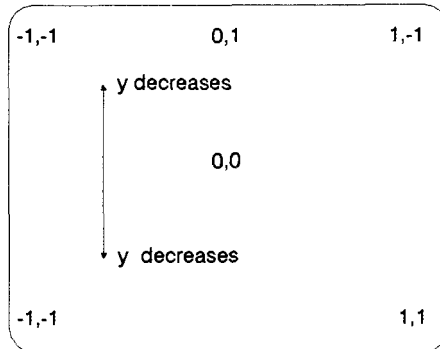
the screen uses the Cartesian coordinates as defined in the following illustration:



If you type:

WINDOW SCREEN (-1,-1)-(1,1)

the screen uses the non-inverted coordinates as defined in the following illustration:



RUN, SCREEN, and WINDOW with no options disable any **WINDOW** definitions and return the screen to its normal physical coordinates.

WRITE Statement

Purpose:

To output data to the screen.

Syntax:

```
WRITE[data][data][...]
```

Comments:

data is the list of numeric and/or string expressions, separated by commas or semicolons, to be output on the screen. If you omit *data*, GW-BASIC outputs a blank line.

The main difference between WRITE and PRINT is that WRITE inserts commas between displayed items and delimits strings with quotation marks. WRITE inserts a carriage return/line feed following the last item in the list.

WRITE outputs numeric values using the same format as PRINT except that it does not precede positive numbers with blank spaces.

Examples:

```
10 A = 80:B = 90:C$ = "THAT'S ALL"  
20 WRITE A,B,C$
```

Displays:

```
80, 90,"THAT'S ALL"
```

WRITE# Statement

Purpose:

To write data to a sequential file.

Syntax:

WRITE #file number, list of expressions

Comments:

file number is the number used to open the file for output.

list of expressions is a list of string and/or numeric expressions, separated by commas or semicolons, to be output to the file.

The **WRITE#** and **PRINT#** statements differ in that **WRITE#** inserts commas between the data items and uses quotation marks to delimit strings. Therefore, you do not need to include explicit delimiters with **WRITE#**. Another difference is that **WRITE#** does not put a blank in front of a positive number.

WRITE# inserts a carriage return/line feed sequence after the last item in the list.

Examples:

```
A$ = "CAMERA": B$ = "93604-1"  
WRITE#1,A$,B$
```

Writes the following image to disk:

```
"CAMERA", "93604-1"
```

A subsequent **INPUT\$** statement, such as the following, inputs "CAMERA" to A\$ and "93604-1" to B\$:

```
INPUT#1,A$,B$
```

Index

Introduction: Since the commands, functions, statements, and variables in the GW-BASIC User's Reference appear in alphabetical order, they are not listed in this index. However, they are listed in the GW-BASIC User's Reference Contents pages. You can also locate them by looking in the index under the function they are related to, such as graphics, strings, and so on.

Please note: Page references for the GW-BASIC User's Guide appear in normal Roman type; page references for the GW-BASIC User's Reference appear in boldface Roman type.

A

absolute value, returning with
ABS, **2**

addition (+) operator, 17, 53

AND logical operator, 56

angles, creating, with CIRCLE,
21

animation. *See* graphics

arctangent of expressions,
returning with ATN, **4**

arithmetic operators

algebraic and BASIC
expressions, 53

chart of, 17, 53

integer division and modulus
arithmetic, 54

arithmetic operators (*Continued*)

order of precedence, 53

overflow and division by
zero, 54-55

array variables, 49

arrays

addresses, returning with
VARPTR, **257-259**

erasing, with ERASE, **65**

maximum values for (DIM),
52

minimum values for
subscripts (OPTION
BASE), **159**

arrow keys, 21

ASCII codes, 69-73

ASC function and, **3**

converting to characters, with
CHR\$, **19**

ASCII codes (*Continued*)
 returning value of, with
 SCREEN, 218
ASCII format, saving files in,
 217
assembly language programs, 77.
 See also programs
 calling subroutines
 with CALL, 11-14, 79-82
 with USR, 83-85, 254
 loading with BLOAD, 8, 78
 memory allocation, 77-78
 programs calling, 85-88
 saving with BSAVE, 8
 starting address for (DEF
 USR), 49
asynchronous communications.
 See communications

B

BACKSPACE key, 21, 23, 24
.BAS extension, 7-8
BASIC programming,
 bibliography for, 3
BASIC programs, converting to
 GW-BASIC, 89-90
BASICA, loading GW-BASIC
 with, 6
bibliography for BASIC
 programming, 3
binary format, saving files in,
 217
block size, maximum, 9
boxes and lines, drawing, with
 LINE, 111-114

buffer, random file
 allocating space for, with
 FIELD, 72

C

calculations
 ? for entering, 17
 distinguished from programs,
 19
 performing, in direct mode,
 17
calling assembly language
 subroutines
 with CALL, 11-14, 79-82
 with USR, 83-85, 254
character device drivers
 IOCTL\$ function and, 102
 IOCTL statement and, 101
characters. *See also* strings
 ASCII character codes,
 69-73
 converting ASCII codes to,
 with CHR\$, 19
 extended codes, 74-75
 returning, with INKEY\$,
 89-90
 INKEY\$ variable and, 89-90
 recognized by GW-BASIC,
 107-108
 returning ASCII code for,
 with SCREEN, 218
 type declaration characters,
 48
child process, 227
circles, creating, 21-22

- clearing
 - display screen, with CLS,
26
 - memory, with NEW, 23, **139**
 - points, with PRESET, **183**
 - variables, with CLEAR, **23**
- closing files
 - with CLEAR, **23**
 - with END, **58**
 - with RESET, **208**
- colon (:), separating statements with, **14**
- colors. *See also* graphics
 - attributes and color ranges for monitors, **167, 224**
 - changing, with PALETTE and PALETTE USING, **165-168**
 - displaying a point, with PSET, **194**
 - examining, with POINT, **179**
 - foreground colors, for screen modes, **225**
 - PAINT statement and, **163-164**
 - screen modes for, **28-30**
 - selecting, with COLOR, **28-30**
- COM trapping, with ON, **143**
- command level, definition, **7**
- command line
 - examples, 9-10
 - filename parameter, 7-8
 - parameter sequence for, **7**
 - < stdin parameter, **8**
 - > stdout parameter, **8**
 - switches in, 8-10
- commands
 - definition, **12**
 - program file commands, **31**
- communications
 - /c switch and, **9**
 - allocating buffer, with OPEN "COM, **156-158**
 - COM trapping, **143**
 - COM(n) statement and, **31**
 - errors, 92-93
 - GET and PUT statements, 94-96
 - INPUT\$ function, **93**
 - input/output functions, **92**
 - input/output statements, **91**
 - opening files, **91**
 - reading files, with INPUT\$ function, **96**
 - TTY sample program, 97-99
- conditional results, IF statement and, **87-88**
- constants
 - definition, **45**
 - numeric constants, 46-47
 - reading, with DATA, **39-40**
 - string constants, **45**
 - type conversion, 51-52
- continuing programs
 - with CONT, **33**
 - with RESUME, **210**
- conventions, notational, **2**
- correcting errors, **23**
- cosine, calculating, **34**
- CTRL-[, **27**
- CTRL-] , **26**
- CTRL-6, **26**
- CTRL--, **25**
- CTRL-\ , **26**

CTRL-B, 25
CTRL-BACKSPACE, 26
CTRL-BREAK, 10, 25
CTRL-C, 25
CTRL-E, 24, 26
CTRL-END, 24, 26
CTRL-ENTER, 14, 27
CTRL-F, 25
CTRL-G, 27
CTRL-H, 23, 25
CTRL-HOME, 27
CTRL-I, 28
CTRL-J, 27
CTRL-K, 27
CTRL-L, 27
CTRL-LEFT ARROW, 25
CTRL-M, 27
CTRL-N, 27
CTRL-NUM LOCK, 28
CTRL-PRINT SCRN, 28
CTRL-R, 27
CTRL-RIGHT ARROW, 25
CTRL-S, 28
CTRL-Z, 10-11

cursor

- finding current position
 - with CSRLIN, 36
 - with LOC, 123
- moving, with LOCATE, 124-125
- positioning, with POS, 182

cursor keys, 21

D

data buffer, number of bytes
for, 8

data files

- overview, 32
- random access files, 37-43
- sequential files, 32-36

data input

- from communications files,
 - with INPUT\$, 96
- from keyboard
 - with INPUT\$, 96
 - with LINE INPUT, 115
- from sequential files
 - with INPUT#, 95
 - with LINE INPUT#, 117
- from terminal, with INPUT, 92-94

DATA statements

- reading, with READ, 202-203
- rereading, with RESTORE, 209

date, setting and retrieving,
with DATE\$, 41

decimal numbers

- converting to hexadecimal numbers, 86
- converting to octal numbers, 140

degrees, converting to radians, 4

DEL key, 21, 23, 24, 26

deleting

- files, with KILL, 31, 107
- lines, with DELETE, 51
- programs, with NEW, 23, 139

device drivers, character

- IOCTL\$ function and, 102
- IOCTL statement and, 101

device errors, finding with

- ERDEV and ERDEV\$, 66

- devices
 - communicating with, via INP, **91**
 - opening input/output to, with OPEN, **150-157**
 - saving memory on, with BSAVE, **10**
 - terminating input/output to, with CLOSE, **25**
 - direct mode of operation, **7**
 - examples of using, **17-18**
 - directories
 - changing, with CHDIR, **18**
 - creating, with MKDIR, **136**
 - removing, with RMDIR, **213**
 - writing directory information, with RESET, **208**
 - display screen
 - arguments, **220**
 - attributes and color ranges for, **167, 224**
 - CGA with color monitor, **219**
 - clearing, with CLS, **26**
 - creating viewports, with VIEW, **261**
 - default attributes, screen mode 10, **222**
 - default foreground colors, **225**
 - EGA with color monitor, **219**
 - EGA with enhanced color monitor, **219**
 - EGA with enhanced display, **220**
 - EGA with monochrome display, **220**
 - line width, setting, with WIDTH, **267-268**
 - display screen (*Continued*)
 - monochrome monitors, **219**
 - printing, **28**
 - redefining coordinates, with WINDOW, **269-271**
 - screen modes
 - for colors, **28-30**
 - for hardware, **220-221**
 - screen pages, **222-223**
 - setting boundaries, with VIEW PRINT, **263**
 - division (/) operator, **17, 53**
 - division by zero, **54-55**
 - double-precision number, converting numeric expression to, **15**
 - double-precision numeric constants, **47**
 - type conversion, **52**
 - double-precision results, /d switch and, **9**
 - double-precision variables, declaring, with DEFDBL, **45**
 - DOWN ARROW, **25**
 - drawing. *See* graphics
- E**
- editing lines, **57**
 - correcting errors, **23**
 - ENTER, for saving edited lines, **24**
 - keys for, **21, 23**
 - methods for, **20, 24**
 - truncating lines with CTRL-END, **24**

ellipses, creating, with
 CIRCLE, **21**
END key, **27**
end of file, finding, **64**
ENTER key, **27**
 as end of statements, **14**
 saving edited lines with, **24**
environment string table
 modifying, with ENVIRON,
 59-60
 retrieving strings from,
 with ENVIRON\$, **61-63**
equality (=) operator, **55, 59**
EQV logical operator, **56**
errors
 communications errors,
 92-93
 correcting, **23**
 device errors, finding with
 ERDEV and ERDEV\$,
 66
 error codes and messages,
 61-67
 error trapping, **147**
 extended error information
 (EXTERR), **71**
 returning error codes with
 ERR and ERL, **67**
 simulating and defining,
 with ERROR, **68-69**
ESC key, **23, 27**
event trapping, with ON, **141-146**
 valid values for, **142**
execution of programs. *See*
 programs
exiting to MS-DOS, **15**
exponentiation, **70**
 ^ operator, **53**

expressions
 assigning value to variable,
 with LET, **110**
 definition, **52**
 truncating to whole numbers,
 with INT, **100**
 type conversion, **51**
extended characters, INKEY\$
 variable and, **89-90**
extended codes, **74-75**

F

FAC, **84**
FCB. *See* File Control Block
File Control Block (FCB)
 number of bytes for, **8**
 returning addresses with
 VARPTR, **257-259**
filename parameter, command
 line, **7-8**
files
 closing
 with CLEAR, **23**
 with END, **58**
 with RESET, **208**
 communications, reading
 with INPUT\$, **96**
 current position, finding
 with LOC, **123**
 data files, **32**
 deleting, with KILL, **31, 107**
 end of file, finding, with
 (EOF), **64**
 input/output, establishing,
 150-155

files (*Continued*)

- length, determining with
 LOF, **128**
- loading, with LOAD, **31, 122**
- locking, with LOCK, **126-127**
- merging, with MERGE, **31, 133**
- printing names of, with
 FILES, **73**
- random access files, **37-43**
- reading records from, with
 GET, **80**
- renaming, with NAME, **138**
- saving, **22**
 - with SAVE, **217**
- sequential, **32-36**
 - reading, with INPUT#,
 95
 - writing data to, with
 WRITE#, **273**
- setting maximum number of
 open files, **8**
- space allocation for, **8**
- terminating input/output to,
 with CLOSE, **25**
- types of, **31**
- unlocking, with UNLOCK,
 251-253
- fixed-point numeric constant, **46**
- floating-point accumulator
 (FAC), **84**
- floating-point division (/)
 operator, **53**
- floating point numeric
 constant, **46**
 - type conversion, **52**
- FOR-NEXT loops, converting
 from BASIC to
 GW-BASIC, **89-90**

- formatted printing, with PRINT
 USING, **186-190**
- function keys, **19, 21** *See also*
 keys; special keys
 - default assignments for, **21, 104**
 - definition, **29**
 - examples, **20**
 - KEY trapping, **143**
 - redefining, **29**
 - with KEY, **103-105**
- functional operators, **58**
- functions
 - communications I/O
 functions, **92**
 - numeric functions, **12**
 - string functions, **13**
 - user-defined, **13**
 - defining and naming, with
 DEF FN, **43-44**

G

- garbage collection, with FRE,
 79
- GML (Graphics Macro
 Language), **53**
- graphics. *See also* colors;
 display screen
 - animation, with GET and
 PUT, **81-82, 197-199**
 - boxes and lines, drawing,
 with LINE, **111-114**
- CIRCLE statement and, **21-22**
- clearing a point, with
 PRESET, **183**

graphics (*Continued*)

- coordinate mapping, with
 PMAP, 178
 - coordinates, retrieving with
 POINT, 179
 - displaying a point, with
 PSET, 183
 - DRAW** statement and, 53-56
 - filling, with **PAINT**, 161-164
 - loading images, with
 BLOAD, 8
 - saving images, with **BSAVE**,
 8
 - transferring images
 - with **GET**, 81-82
 - with **PUT**, 197
 - windows, creating with
 VIEW, 261
- Graphics Macro Language, 53
- greater than (**>**) operator, 55,
 59
- greater than or equal to (**>=**)
 operator, 55, 59
- GW-BASIC**, converting **BASIC**
 programs to, 89-90

H

- hexadecimal equivalents, 101-
 103
- hexadecimal numbers, converting
 decimal values
 - to, 86
- hexadecimal numeric constant,
 46
- HOME** key, 27

I

- images. *See* graphics
- IMP** logical operator, 56
- indirect mode of operation, 7
 - examples of using, 18-19
- inequality (**< >**) operator, 55,
 59
- input. *See* data input
- input/output
 - OPEN** statement for, 150-157
 - terminating, with **CLOSE**, 25
- INS** key, 21, 24, 27-28
- integer division and modulus
 arithmetic, 54
- integer numeric constant, 46
- integer variables, declaring,
 with **DEFINT**, 45

J

- joysticks
 - coordinates, returning with
 STICK, 236
 - status of buttons,
 - returning, with **STRIG**,
 239-240
 - STRIG** trapping, 145
 - trapping buttons, with
 STRIG(number), 241

K

- key capture, with **KEY(number)**,
 106
- key scan codes, 105-106

KEY trapping, with ON, 143-144

keyboard buffer, reading, with
INKEY\$, 89-90

keys. *See also* function keys;

special keys

CTRL-Z, 10-11

cursor keys, 21

editing keys, 23

summary chart of editing
keys, 21

keywords, definition, 11-12

L

LEFT ARROW, 26

left-justifying data, with

LSET, 132

length of files, determining

with **LOF, 128**

less than (**<**) operator, 55, 59

less than or equal to (**<=**)

operator, 55, 59

light pen

reading, with **PEN** statement,
173

reading coordinates, with

PEN function, **171-172**

line format, 13, 14 *See also*

statements

CTRL-ENTER, for wrapping

lines, 14

ENTER, for ending lines, 14

line numbers, 14

wrapping of lines, 14

lines

branching

with **GOTO, 85**

with **ON/GOSUB, 149**

with **ON/GOTO, 149**

deleting, with **DELETE, 51**

editing, with **EDIT, 57**

inputting, with **LINE INPUT,**
115

numbering, with **AUTO**

command, **5**

renumbering, with **RENUM,**
206

sequential file input, with

LINE INPUT#, 117

lines and boxes, drawing, with

LINE, 111-114

loading

assembly language programs,

with **BLOAD, 8, 78**

files, with **LOAD, 31, 122**

graphic images, with

BLOAD, 8

GW-BASIC, 5

with **BASICA, 6**

locking files

with **LOCK, 126-127**

with **OPEN, 153-155**

logarithms, calculating, **129**

logical operators

chart of, 56-57

definition, 56

example, 58

type conversion, 51

uses for, 57

loops
 FOR and NEXT statements
 and, **76-78**
 WHILE-WEND statement
 and, **265**

M

machine language programs. *See*
 assembly language
 programs
masks, tile. *See* tile mask
masks for printing. *See*
 formatted printing
MAT functions, converting from
 BASIC to GW-BASIC,
 89-90
memory
 assembly language
 subroutines allocation,
 77-78
 clearing, with NEW, 23, **139**
 garbage collection, with
 FRE, **79**
 highest location, setting, 9
 reading, with PEEK, **170**
 requirements, for storing
 variables, 50
 saving on devices, with
 BSAVE, **10**
 writing to, with POKE, **181**
merging program files, 31, **133**
messages and error codes,
 61-67
MOD operator, 54

modes of operation. *See* direct
 mode of operation;
 indirect mode of
 operation
modifying program lines. *See*
 editing lines
modulus arithmetic, 54
monitors. *See* display screen
movement commands, for DRAW
statement, **53-55**
MS-DOS. *See* operating system
multiple assignments or
 statements, converting
 from BASIC to
 GW-BASIC, 89-90
multiplication (*) operator,
 17, 53
music. *See also* sound
 generation
 PLAY function and, **177**
 PLAY statement and, **174-176**
 PLAY trapping, **144**

N

NAME command, 31-32
negation (-) operator, 53
nested loops, FOR and NEXT
 statements and, 77
networks
 LOCK statement and, **126-127**
 OPEN statement and, **153-155**
 UNLOCK statement and,
 251-253
NOT logical operator, 56

notational conventions, 2
NTF values, 83
number type flag (NTF values), 83
numbers
 cosine calculation, 34
 decimal
 converting to hexadecimal (HEX\$), 86
 converting to octal (OCT\$), 140
 double-precision, converting (CDBL), 15
 logarithm calculation, 129
 rounding, with CINT, 20
 sign of, returning (SGN), 226
 sine calculation, 229
 single-precision, converting (CSNG), 35
 square root calculation, 235
 strings, converting (CVI, CVS, CVD), 37
 tangent calculation, 246
 truncating
 with FIX, 75
 with INT, 100
numeric constants
 reading, with DATA, 39-40
 single- and double-precision form, 47
 types of, 46
numeric functions, definition, 12
numeric variables, setting to zero, with CLEAR, 23

O

octal numbers, converting from decimal values (OCT\$), 140
octal numeric constant, 46
Ok prompt, 7
operating system
 requirements, 1
 returning to, 15
 with SYSTEM, 244
operators
 arithmetic operators, 17, 53-55
 definition, 52
 functional, 58
 logical, 56-58
 relational, 55
 string, 59-60
OR logical operator, 56
output. *See* input/output
output ports, sending data bytes to, 160
overflow and division by zero, 54-55
overlays
 MERGE command and, 16-17
 transferring control to, with CHAIN, 16-17

P

pages, copying, with PCOPY, 169
painting graphics. *See* graphics
palette. *See* colors

- parameters, sequence of, in
 - command line, 7
- pen. *See* light pen
- PLAY trapping, 144. *See also* music
- points. *See* graphics
- ports
 - INP function and, 91
 - sending data bytes to, 160
 - WAIT statement and, 264
- print head, current position of (LPOS), 130
- printing
 - to line printer, with LPRINT and LPRINT USING, 131
 - line width, setting, with WIDTH, 267-268
 - programs
 - with LIST, 19, 119
 - with LLIST, 121
 - on the screen
 - with PRINT, 184
 - with PRINT USING, 186-190
 - with WRITE, 272
 - screen, printing, 28
 - to sequential files, with PRINT# and PRINT# USING, 191-193
 - spacing
 - with SPC, 234
 - with TAB, 245
- program file commands, 31
- programs. *See also* assembly language programs; subroutines
 - branching
 - with ON, 141-146
 - with GOTO, 85
 - programs (*Continued*)
 - with ON/GOTO, 149
 - with ON/GOTO, 149
 - continuing
 - with CONT, 33
 - with RESUME, 210
 - deleting, with NEW, 23, 139
 - distinguished from calculations, 19
 - executing
 - another program, with SHELL, 227
 - with RUN, 18, 19, 31, 216
 - flow of, controlled with IF, 87-88
 - listing
 - with LIST, 19, 119
 - with LLIST, 121
 - merging, 133
 - passing variables to, with COMMON, 32
 - printing
 - with LIST, 19, 119
 - with LLIST, 121
 - suspending execution, with WAIT, 264
 - terminating
 - with END, 58
 - with STOP, 237
 - tracing execution, with TRON/TROFF, 250
 - transferring control to, with CHAIN and MERGE, 16-17

Q

- question mark (?), for entering calculations, 17

R

radians, converting from
degrees, **4**

random access files
accessing, 39-43
creating, 37-38
modes for opening, **151**

random file buffer, allocating
space, with FIELD, **72**

random numbers
getting, with RND, **214**
reseeding with
RANDOMIZE, **200-201**

reading values from DATA
statements
with READ, **202-203**
rereading, with RESTORE,
209

records
maximum length, **8**
reading, with GET, **80**
writing to random disk file,
with PUT, **196**

redirection of input and
output, 10-11

relational operators, **55**

remarks, inserting with REM,
204

renaming files, with NAME, **138**

reserved words, *See* keywords

RETURN, event trapping and,
143

RIGHT ARROW, 26, 28

right-justifying data, with
RSET, **215**

rounding numbers, with CINT,
20

RS-232 port. *See*
communications

S

saving
assembly language programs,
with BSAVE, **8**
files, with SAVE, 22, 31, **217**
images, with BSAVE, **8**
memory on devices, with
BSAVE, **10**

scan codes, key, 105-106

screen. *See* display screen

screen pages
copying, with PCOPY, **169**
managing, with SCREEN,
222-223

segments, assigning current
address, with DEF SEG,
47

sequential files
accessing, 35
adding data to, 36
creating, 32-34
modes for opening, **150**
writing data to, with
WRITE#, **273**

serial communications. *See*
communications

sharing files. *See* locking
files

SHIFT-PRINT SCRNL, 28

single-precision numbers,
converting numeric
expression to, **35**

- single-precision numeric constants, 47
- single-precision variables, declaring, with DEFSNG, 45
- sound generation, **230**. *See also* music
 - chart of frequencies, **231**
 - clock ticks for tempos, **232**
 - sounding the speaker, with BEEP, 7
- spaces
 - adding, with SPACE\$, **233**
 - skipping
 - with SPC, **234**
 - with TAB, **245**
- special keys
 - BACKSPACE, 21, 24, 25
 - CTRL-[, 27
 - CTRL-], 26
 - CTRL-6, 26
 - CTRL--, 25
 - CTRL-\, 26
 - CTRL-B, 25
 - CTRL-BACKSPACE, 26
 - CTRL-BREAK, 10, 25
 - CTRL-C, 25
 - CTRL-E, 24, 26
 - CTRL-END, 24, 26
 - CTRL-ENTER, 14, 27
 - CTRL-F, 25
 - CTRL-G, 27
 - CTRL-H, 23, 25
 - CTRL-HOME, 27
 - CTRL-I, 28
 - CTRL-J, 27
 - CTRL-K, 27
 - CTRL-L, 27
 - CTRL-LEFT ARROW, 25
 - special keys (*Continued*)
 - CTRL-M, 27
 - CTRL-N, 27
 - CTRL-NUM LOCK, 28
 - CTRL-PRINT SCRNL, 28
 - CTRL-R, 27
 - CTRL-RIGHT ARROW, 25
 - CTRL-S, 28
 - DEL, 21, 23, 24, 26
 - DOWN ARROW, 25
 - END, 27
 - ENTER, 14, 24, 27
 - ESC, 23, 27
 - HOME, 27
 - INS, 21, 24, 27-28
 - LEFT ARROW, 26
 - RIGHT ARROW, 26, 28
 - SHIFT-PRINT SCRNL, 28
 - TAB, 28
 - UP ARROW, 26
 - square roots, calculating, **235**
 - stacks, CALL statement and, 79-82
 - starting GW-BASIC, 5 *See* loading GW-BASIC
 - statements. *See also* line
 - format
 - definition, 12, 13
 - more than one per line, 14
 - processing of, 14-15
 - < stdin parameter, command line, 8
 - > stdout parameter, command line, 8
 - stopping. *See* terminating
 - STRIG trapping, **145**. *See also* joysticks

- string constants, 45
 - reading, with DATA, 39-40
 - string functions, definition, 13
 - string memory, garbage
 - collection, with FRE, 79
 - string operators, 59-60
 - string variables
 - allocating space for, with FIELD, 72
 - declaring, with DEFSTR, 45
 - setting to null, with CLEAR, 23
 - strings. *See also* characters
 - adding spaces, with SPACE\$, 233
 - ASCII codes for, 3
 - counting characters in, with LEN, 109
 - creating, with STRING\$, 242
 - dimensions, converting from BASIC to GW-BASIC, 89-90
 - leftmost portion, returning with LEFT\$, 108
 - numbers, converting to strings, with STR\$, 238
 - numeric values
 - converting to, with MKI\$, MKS\$, MKD\$, 137
 - returning, with VAL, 256
 - replacing, with MID\$, 135
 - rightmost portion, returning with RIGHT\$, 212
 - strings (*Continued*)
 - searching for, with INSTR, 98
 - substrings, returning, with MID\$, 134
 - subdirectories. *See* directories
 - subroutines. *See also* assembly language programs; programs
 - calling assembly language subroutines, 11-14
 - GOSUB and RETURN statements for, 83-84
 - returning from, with RETURN, 211
 - substrings, returning, with MID\$, 134
 - subtraction (-) operator, 17, 53
 - switches
 - /c, 9
 - /d, 9
 - /f, 8
 - /i, 8
 - /m, 9
 - /s, 8
 - system requirements, 1
- T**
- TAB key, 28
 - tabs, setting, 245
 - terminating
 - input/output, with CLOSE, 25
 - programs
 - with END, 58
 - with STOP, 237
 - tile mask, PAINT statement 161-164

time

counting seconds, with

TIMER, 249

setting or retrieving, with

TIME\$, 247-248

TIMER trapping, 145

tracing execution of statements

(**TRON/TROFF**), **250**

trapping events, with **ON**,

141-146

COM trapping, 143

KEY trapping, 143-144

PLAY trapping, 144

STRIG trapping, 145

TIMER trapping, 145

valid value for, **142**

TTY sample communication

program, **97-99**

type conversion, **51-52**

type declaration characters,

48

U

unlocking files, with **UNLOCK**,

251-253

UP ARROW, 26

user-defined functions

defining and naming, with

DEF FN, 43-44

definition, **13**

V

variables

addresses, returning, with

VARPTR, 257-259

allocating space for, with

FIELD, 72

array variables, **49**

assigning value from

expressions, with **LET**,

110

character form of offset,

VARPTR\$ and, **260**

clearing, **23**

declaring types, with

DEFINT/SNG/DBL/

STR, 45

definition, **13, 47**

exchanging values, with

SWAP, 243

memory requirements for, **50**

names, **48**

passing to chained programs,

with **COMMON, 32**

type declaration characters,

48

viewports

creating, with **VIEW, 261**

redefining coordinates, with

WINDOW, 269-271

setting boundaries, with

VIEW PRINT, 263

W

windows

creating, with VIEW, 261

redefining coordinates, with

WINDOW, 269-271

setting boundaries, with

VIEW PRINT, 263

working directories. *See*
directories

X

XOR logical operator, 56

Z

zero, division by, 54-55

