

TeleVideo® GWBASIC User's Manual

TeleVideo®
GWBASICSTM User's Manual

TeleVideo Part Number 125681-00 Rev. A1

January 1984

Copyright (c) 1984 by TeleVideo Systems, Inc. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without prior written permission of TeleVideo Systems, Inc., 1170 Morse Avenue, P.O. Box 3568, Sunnyvale, California 94088.

Disclaimer

TeleVideo Systems, Inc. makes no representations or warranties with respect to this manual. Further, TeleVideo Systems, Inc. reserves the right to make changes in the specifications of the product described within this manual at any time without notice and without obligation of TeleVideo Systems, Inc. to notify any person of such revision or changes.

TeleVideo is a registered trademark of TeleVideo Systems, Inc.
TeleDOSTM is a trademark of TeleVideo Systems, Inc.
GWBASICS is a trademark of TeleVideo Systems, Inc.

TeleVideo Systems, Inc.
1170 Morse Avenue
P.O. Box 3568
Sunnyvale, CA 94088
408/745-7760



TABLE OF CONTENTS

1.	INTRODUCTION	1.1
	How To Use This Manual	1.1
	Notation Conventions	1.2
2.	USING GWBASIC	2.1
	Starting GWBASIC	2.1
	Command Line Options	2.1
	Redirection of Standard Input and Output	2.5
	Redirection of the Standard Input	2.5
	Redirection of the Standard Output	2.5
	Examples of Redirecting the Standard Input and Output	2.6
	Modes of Operation	2.6
	Direct Mode	2.7
	Indirect Mode	2.7
	Keyboard Usage	2.7
	Main Keyboard	2.8
	Numeric Keypad	2.10
	Special Key Combinations	2.11
	GWBasic Program Editor	2.11
	Program Editor Keys	2.12
	Entering Text Using the Program Editor	2.15
	Changing Characters on the Current Line	2.16
	Deleting Characters on the Current Line	2.16
	Adding Characters to the Current Line	2.17
	Deleting a Portion of the Current Line	2.17
	Cancelling the Current Line	2.18
	Entering or Changing a BASIC Program	2.18
	Adding a New Program Line	2.18
	Changing an Existing Program Line	2.19
	Deleting a Program Line	2.20
	Syntax Errors	2.21
3.	GENERAL PROGRAMMING INFORMATION	3.1
	Line Format	3.1
	Character Set	3.2
	Constants	3.3
	Precision for Numeric Constants	3.4
	Variables	3.5
	Variable Names	3.5
	Declaring Variable Types	3.6
	Array Variables	3.7
	Numeric Type Conversion	3.8

GWBasic User's Manual

Expressions and Operators	3.9
Arithmetic Operators	3.10
Integer Division and Modulus Arithmetic	3.11
Relational Operators	3.12
Logical Operators	3.13
How Logical Operators Work	3.15
Functional Operators	3.16
Order of Execution	3.16
String Operators	3.16
Concatenation	3.17
String Functions	3.17
4. INPUT AND OUTPUT	4.1
Display Screen I/O	4.1
Text Modes	4.1
Graphics Modes	4.3
Medium Resolution	4.4
High Resolution	4.4
Specifying Graphics Coordinates	4.4
Files	4.6
Naming Files	4.6
Device Names	4.6
Filenames	4.7
Tree-Structured Directories	4.8
Commands for Program Files	4.9
Protected Files	4.10
File Numbers	4.10
Disk Data Files	4.11
Sequential Files	4.11
Changing a Sequential File	4.13
Random Access Files	4.13
Special I/O Features	4.17
5. BASIC COMMANDS, STATEMENTS, AND FUNCTIONS	5.1
ABS	5.3
ASC	5.4
ATN	5.5
AUTO	5.6
BEEP	5.8
BLOAD	5.9
BSAVE	5.10
CALL	5.11
CDBL	5.12
CHAIN	5.13
CHDIR	5.15
CHR\$	5.16
CINT	5.17
CIRCLE	5.18
CLEAR	5.21
CLOSE	5.23
CLS	5.24
COLOR (Text)	5.25
COLOR (Graphics)	5.27

GWBASIC User's Manual

COM(n)	5.28
COMMON	5.29
CONT	5.30
COS	5.31
CSNG	5.32
CSRLIN	5.33
CVI, CVS, CVD	5.34
DATA	5.35
DATE\$ (Function)	5.36
DATE\$ (Statement)	5.37
DEF FN	5.38
DEF SEG	5.40
DEFtype	5.42
DEF USR	5.43
DELETE	5.44
DIM	5.45
DRAW	5.47
EDIT	5.51
END	5.52
EOF	5.53
ERASE	5.54
ERR and ERL	5.55
ERROR	5.57
EXP	5.59
FIELD	5.60
FILES	5.62
FIX	5.63
FOR...NEXT	5.64
FRE	5.67
GET (Files)	5.68
GET (Graphics)	5.69
GOSUB...RETURN	5.72
GOTO	5.74
HEX\$	5.75
IF	5.76
INKEY\$	5.78
INP	5.79
INPUT	5.80
INPUT#	5.82
INPUT\$	5.83
INSTR	5.84
INT	5.85
KEY (Key Trapping)	5.86
KEY (Soft Keys)	5.88
KEY(n)	5.90
KILL	5.91
LEFT\$	5.92
LEN	5.93
LET	5.94
LINE	5.95
LINE INPUT	5.97
LINE INPUT#	5.98
LIST	5.99
LLIST	5.101
LOAD	5.102

GWBASIC User's Manual

LOC	5.103
LOCATE	5.104
LOF	5.106
LOG	5.107
LPOS	5.108
LPRINT and LPRINT USING	5.109
LSET and RSET	5.111
MERGE	5.112
MID\$ (Function)	5.113
MID\$ (Statement)	5.114
MKDIR	5.115
MKI\$, MKS\$, MKD\$	5.116
NAME	5.117
NEW	5.118
OCT\$	5.119
ON COM(n)	5.120
ON ERROR	5.122
ON...GOSUB and ON...GOTO	5.124
ON KEY(n)	5.125
ON PEN	5.127
ON PLAY(n)	5.129
ON STRIG(n)	5.131
ON TIMER	5.133
OPEN	5.135
OPEN COM	5.137
OPTION BASE	5.140
OUT	5.141
PAINT	5.142
PEEK	5.146
PEN (Function)	5.147
PEN (Statement)	5.149
PLAY (Function)	5.150
PLAY (statement)	5.151
PLAY ON, PLAY OFF, PLAY STOP	5.155
PMAP	5.156
POINT (Function)	5.157
POINT (Statement)	5.158
POKE	5.159
POS	5.160
PRESET	5.161
PRINT	5.162
PRINT USING	5.164
PRINT# and PRINT# USING	5.169
PSET	5.172
PUT (Files)	5.173
PUT (Graphics)	5.174
RANDOMIZE	5.177
READ	5.178
REM	5.179
RENUM	5.180
RESET	5.181
RESTORE	5.182
RESUME	5.183
RETURN	5.184
RIGHT\$	5.185

GW BASIC User's Manual

RMDIR	5.186
RND	5.187
RUN	5.188
SAVE	5.189
SCREEN (Function)	5.190
SCREEN (Statement)	5.191
SGN	5.193
SIN	5.194
SOUND	5.195
SPACE\$	5.197
SPC	5.198
SQR	5.199
STICK(n)	5.200
STOP	5.201
STR\$	5.202
STRIG	5.203
STRIG(n)	5.205
STRING\$	5.206
SWAP	5.207
SYSTEM	5.208
TAB	5.209
TAN	5.210
TIME\$ (Function)	5.211
TIME\$ (Statement)	5.212
TIMER	5.213
TRON/TROFF	5.214
USR	5.215
VAL	5.216
VARPTR	5.217
VARPTR\$	5.219
VIEW	5.220
VIEW PRINT	5.223
WAIT	5.224
WHILE...WEND	5.225
WIDTH	5.226
WINDOW	5.228
WRITE	5.231
WRITE#	5.232

APPENDICES

A. Error Codes and Error Messages	A.1
B. ASCII Character Codes	B.9
C. Keyboard Scan Codes	C.14
D. GW BASIC Reserved Words	D.15
E. Mathematical Functions	E.16
F. Technical Information	F.17

LIST OF FIGURES

2-1	The Keyboard	2.7
2-2	Main Keyboard	2.8
2-3	Numeric Keypad	2.10
4-1	Text Screen	4.2
4-2	Graphics Screen	4.5

LIST OF TABLES

1-1	Notation Conventions	1.2
2-1	Main Keyboard Special Keys	2.8
2-2	Special Key Combinations	2.11
2-3	Program Editor Keys	2.12
3-1	Special Character Keys	3.2
4-1	Device Names	4.7
4-2	Program File Commands	4.10
4-3	Additional I/O Support Devices	4.17
A-1	Error Message Quick Reference Guide	A.1
A-2	Error Messages	A.2
B-1	ASCII Character Codes	B.10
B-2	Extended ASCII Codes	B.13
E-1	Mathematical Functions	E.16
F-1	GWBASIC FCB	F.20

1. INTRODUCTION

BASIC is an easy-to-learn, easy-to-use, high-level programming language. BASIC stands for Beginner's All-purpose Symbolic Instruction Code.

GWBasic is the BASIC interpreter for TeleVideo's personal computers using the TeleDOS disk operating system. GWBasic is a powerful programming language providing advanced features like color graphics, sound and music, and event trapping.

HOW TO USE THIS MANUAL

This manual is intended as a reference manual to describe the many features of GWBasic. To use this manual effectively, you should have a working knowledge of the BASIC programming language and general programming concepts.

The manual is divided into five chapters plus a number of appendices.

Chapter 1 is an introduction to the GWBasic manual and includes a listing of the syntax notation used throughout the manual.

Chapter 2 tells you how to get started using GWBasic. Included is information on loading the GWBasic interpreter into the computer, the modes of operation, and how to create and edit a BASIC program.

Chapter 3 covers a variety of topics you need to know to program using the GWBasic language. This chapter discusses line format, the GWBasic character set, constants and variables, and expressions and operators.

Chapter 4 discusses input and output (I/O) in GWBasic. This chapter includes sections on using the screen modes, naming files, use of sequential and random disk files, and other special I/O features.

Chapter 5 is a reference section. It contains a listing with detailed descriptions of the GWBasic commands, statements, and functions.

The Appendices include useful reference information, such as a list of error codes and error messages, the extended ASCII character chart, how to derive additional mathematical functions, and a listing of GWBasic's reserved words.

Notation Conventions

Several notation conventions are used throughout the manual to make it easier to describe the syntax for GWBasic's many commands, statements, and functions. Table 1-1 is a listing of these conventions.

**Table 1-1
Notation Conventions**

Symbol	Description
[]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate data you enter. When the angle brackets enclose lower-case text, type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper-case text (or a capital followed by lower-case), you must press the key named by the text; for example, <Ctrl>.
{ }	Braces indicate that you have a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets. The choices are separated by the bar () symbol.
	Vertical bars separate the choices within braces. At least one of the entries separated by bars must be entered unless the entries are also enclosed by square brackets.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.
<CR>	Indicates you are to press the <Enter> key. CR stands for Carriage Return.
^	The caret is often used as a shorthand notation for the control key; therefore, the notation ^Z indicates you are to press the <Ctrl> key and the upper-case Z key at the same time.
/	This symbol, when used between two or more keys, indicates that you are to press the keys simultaneously. For example, <Ctrl>/<Break> indicates you are to press the <Ctrl> and <Break> keys at the same time.
CAPS	Capital letters indicate portions of statements or commands that must be entered exactly as shown.

All punctuation (except those notation items listed above) must be entered as shown in the format for the statement or command.

2. USING GWBasic

This chapter tells how to start GWBasic. It then explains how to use the GWBasic editor to write and edit programs.

STARTING GWBasic

Your GWBasic interpreter comes on your TeleDOS operating diskette. The interpreter resides in the file labeled BASIC.EXE and is started similar to other utility or applications programs. To start BASIC:

1. Turn on your computer.
2. Insert the TeleDOS system diskette in drive A and load TeleDOS into the computer.
3. Enter the following command in response to the A> system prompt:

BASIC<CR>

GWBasic displays an identifying sign-on message along with the number of free bytes in memory for you to use. GWBasic then displays the **Ok prompt**. This indicates that GWBasic is at the **command level** and is waiting for a command.

Command Line Options

GWBasic allows optional parameters to be included on the command line when GWBasic is started. These parameters allow the environment in which GWBasic operates to be altered, or requests GWBasic to immediately load and run a BASIC program.

GWBasic will work without these options. If you are new to BASIC, you may wish to skip over this section for now. You can refer back to this section when you become more familiar with the BASIC language and the capabilities of GWBasic.

The complete format of the GWBasic command line is as follows:

```
BASIC [<filespec>]
      [<standard input>[[>]>standard output]
      [/F:<# of files>]
      [/S:<record size>]
      [/C:<buffer size>]
      [/D]
      [/I]
      [/M:[<maximum workspace>][,<maximum block size>]]
```

- `filespec` This is the name or file specification of a BASIC program to be loaded and run immediately. If the file is found, GWBasic proceeds as if a `RUN <filespec>` command were given in response to the `Ok` prompt. `<filespec>` must conform to the rules for naming files as described in Chapter 4 and may contain a path. A default file extension of `.BAS` is used if none is included. The file specification should not be enclosed in quotation marks.
- `<standard input` A GWBasic program normally receives its input from the keyboard (the standard input device). This option allows the input to be received from a specified file. When you use the `<standard input` option, it must be positioned before any of the slash (`/`) type parameters. Refer to the following section on Redirection of Standard Input and Output for more information.
- `[>]>standard output` A GWBasic program normally sends its output to the screen (the standard output device). This option allows the output to be sent to a specified file or device. When you use the `>standard output` option, it must be positioned before any of the slash (`/`) type parameters. Refer to the following section on Redirection of Standard Input and Output for more information.
- `/F:<# of files>` This parameter is ignored unless the `/I` parameter is also specified.
- This parameter sets the number of files that may be open simultaneously during execution of a BASIC program. If omitted, the default value is five files.
- The maximum number of files supported by GWBasic is 15. The actual number of files that can be open simultaneously is limited by the value of the `FILES=` parameter in the `CONFIG.SYS` file. If `FILES=` is not specified in the `CONFIG.SYS` file, the default value is 8. In this case, since GWBasic uses three files by default, you are limited to five additional files (`/F:5`).
- Each open file requires 188 bytes of memory for the file control block (FCB), plus the record length for random access files.

- `/S:<record size>` This parameter is ignored unless the `/I` parameter is also specified.
- This parameter sets the maximum record size in bytes used for random access files. If this parameter is included, the record length parameter of the OPEN statement cannot exceed this record size value. The maximum value allowed is 32767 bytes.
- `/C:<buffer size>` This parameter sets the buffer size in bytes for receiving data on the RS-232 asynchronous communications port. The buffer for transmitting data is always set to 128 bytes. If this parameter is omitted, the default value is 256 bytes. A maximum value of 32767 is allowed.
- RS-232 support can be disabled by setting the buffer size to 0 (`/C:0`). In this case, no buffer space is reserved for communications and the communications support is not included when GWBASIC is loaded.
- `/D` This parameter loads the support for the double-precision Transcendental math package. This allows the functions ATN, COS, EXP, LOG, SIN, SQR, and TAN to work with double-precision numbers. This package increases the resident size of GWBASIC by approximately 2,000 bytes.
- `/I` GWBASIC is able to dynamically allocate the space required to support file operations; therefore, the `/S` and `/F` parameters need not be supported. However, certain applications packages have been written so that certain BASIC internal data structures must be static. In order to provide compatibility with these BASIC programs, GWBASIC statically allocates the space required for file operations based on the `/S` and `/F` parameters when the `/I` parameter is also specified.

`/M:[<maximum workspace>][,<maximum block size>]`

GWBASIC is able to use a maximum of 64 Kbytes of memory. When the /M: parameter is omitted, GWBASIC allocates all available memory up to a maximum of 64 Kbytes for a workspace. This workspace includes the interpreter work area, the GWBASIC stack area, and space for your programs and data.

The <maximum workspace> option is used to set the maximum number of bytes used for the GWBASIC workspace. By limiting the GWBASIC workspace to less than 64 Kbytes, you can reserve space for machine language subroutines or for special data storage above the GWBASIC workspace.

The <maximum block size> option is used to set the maximum number of bytes to be used by the GWBASIC workspace and for special user storage. The <maximum block size> is entered as the number of 16 byte paragraphs to reserve. When omitted, the default value is 4096 (16 x 4096 = 64K). By specifying more than 4096 paragraphs, you can reserve user storage space above the GWBASIC workspace without decreasing the size of the GWBASIC workspace.

NOTE! The values for <# of files>, <record size>, <buffer size>, <maximum workspace>, and <maximum block size> may be entered as decimal, octal (preceded by &O), or hexadecimal (preceded by &H) numbers.

Examples:

`BASIC PAYROLL`

GWBASIC is loaded into memory using a 64K maximum workspace and allows five files to be open simultaneously. Once GWBASIC is loaded, the BASIC program PAYROLL.BAS is loaded and run.

`BASIC /C:0 /M:32768`

GWBASIC is loaded into memory with the RS-232 support disabled. Sixty-four Kbytes are reserved for GWBASIC workspace and user program space. The GWBASIC workspace uses the lower 32 Kbytes and the upper 32 Kbytes are available for the user.

BASIC /M:,4112

GWBASIC is loaded into memory with 66048 bytes (16 x 4112 = 66048) allocated for the GWBASIC workspace and user workspace. The GWBASIC workspace uses the lower 64 Kbytes (65536 bytes), with 512 bytes available above GWBASIC for the user.

BASIC /M:32000,2048

GWBASIC is loaded into memory with 32 Kbytes (16 x 2048 = 32768 = 32K) allocated for GWBASIC workspace and user program space. The GWBASIC workspace uses the lower 32000 bytes, leaving 768 bytes available above GWBASIC for the user.

REDIRECTION OF STANDARD INPUT AND OUTPUT

GWBASIC allows you to redirect your BASIC input and output by including a new standard input or output on the GWBASIC command line. The form of the command line is:

```
BASIC [<filespec>] [<standard input> [[>][>standard output]]
```

Redirecting the Standard Input

GWBASIC normally reads input from the keyboard, but can be directed to read input from the standard input file specified on the GWBASIC command line. When the input is redirected, the INPUT, LINE INPUT, INPUT\$, and INKEY\$ statements read from the specified standard input file.

GWBASIC continues to read from the specified standard input file until a ^Z end-of-file character is detected. This condition may be tested using the BASIC EOF function. If the file is not terminated by a ^Z end-of-file character, or a BASIC input statement tries to read beyond the end-of-file character, all open files are closed, the program is terminated, and the error message "Read past end" is sent to the standard output.

File input from "KYBD:" is always read from the keyboard, regardless of the standard input. GWBASIC also continues to trap keys from the keyboard when the ON KEY(n) statement is used.

Redirecting the Standard Output

GWBASIC normally writes output to the screen, but can be directed to write to the standard output file or device specified on the GWBASIC command line. When output is redirected, all PRINT statements write to the specified standard output. Error messages are also sent to the standard output.

If two greater-than (>) symbols are entered in front of the new standard output file, all output sent to the file is appended to the end of the file.

File output to "SCRN:" is always written to the screen, regardless of the standard output.

The <Ctrl>/<PrtSc> echoing to the printer feature is disabled when the standard output is redirected.

Examples of Redirecting the Standard Input and Output

BASIC MYPROG <DATA.IN

GWBASIC is loaded into memory and loads and runs the BASIC program MYPROG.BAS. If MYPROG.BAS requires standard input, the data is read from the file DATA.IN. All standard output from MYPROG.BAS is sent to the screen. The files BASIC.EXE, MYPROG.BAS, and DATA.IN are located on the default drive.

BASIC MYPROG <DATA.IN >B:DATA.OUT

GWBASIC is loaded into memory and loads and runs the BASIC program MYPROG.BAS (BASIC.EXE and MYPROG.BAS are located on the default drive). If MYPROG.BAS requires standard input, the data is read from the file DATA.IN on the default drive. All standard output from MYPROG.BAS is written to file DATA.OUT on drive B.

BASIC SALES >>B:\SALES\SALES.DAT

GWBASIC is loaded into memory and loads and runs the BASIC program SALES.BAS (BASIC.EXE and SALES.BAS are located on the default drive). All standard input to the SALES program comes from the keyboard. All standard output is appended to the file SALES.DAT located in the SALES directory on drive B. (For more information on directories and the use of paths, refer to Chapter 3 of the TeleDOS User's Manual.)

MODES OF OPERATION

Once GWBASIC is started, it displays the Ok prompt. The Ok indicates that GWBASIC is ready and waiting for your command. This state is often referred to as the **command level**. From the command level, you have a choice of two modes of operation: the direct mode or the indirect mode.

Direct Mode

The direct mode is used to send instructions directly to the interpreter. The BASIC statements and commands are entered without being preceded by line numbers. When the <Enter> key is pressed, the statements and commands are executed immediately. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions are lost (not stored in memory) after execution. The direct mode is useful for program debugging and for making quick calculations that do not require a complete program. For example:

```
Ok
PRINT "THE SQUARE ROOT OF 7 IS" SQR(7)    (you enter)
THE SQUARE ROOT OF 7 IS 2.645751         (GWBASIC responds)
Ok
```

Indirect Mode

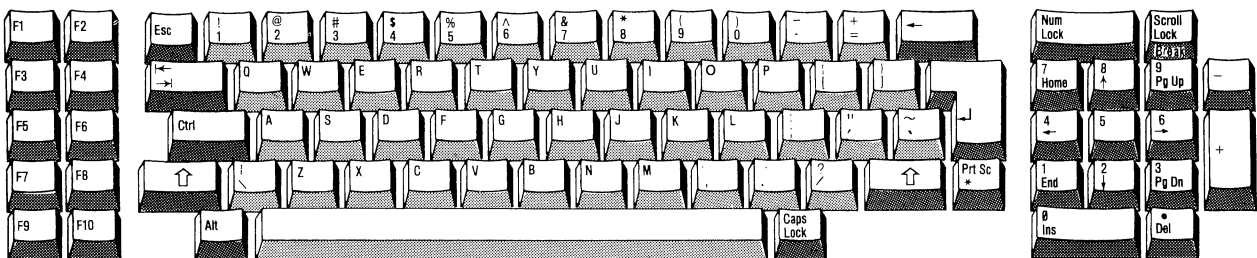
The indirect mode is used to write BASIC programs. To indicate the line is part of a program, you start the line with a line number. When the <Enter> key is pressed, the entered line is stored in memory as part of the program. To execute the program in memory, enter the RUN command. For example:

```
Ok
10 REM THIS PROGRAM CALCULATES THE SQUARE ROOT OF SEVEN
20 PRINT "THE SQUARE ROOT OF 7 IS" SQR(7)
30 END
RUN
THE SQUARE ROOT OF 7 IS 2.645751
Ok
```

KEYBOARD USAGE

The keyboards on TeleVideo personal computers using GWBASIC are divided into three sections: the **main keyboard**, the **numeric keypad**, and the **function keys**.

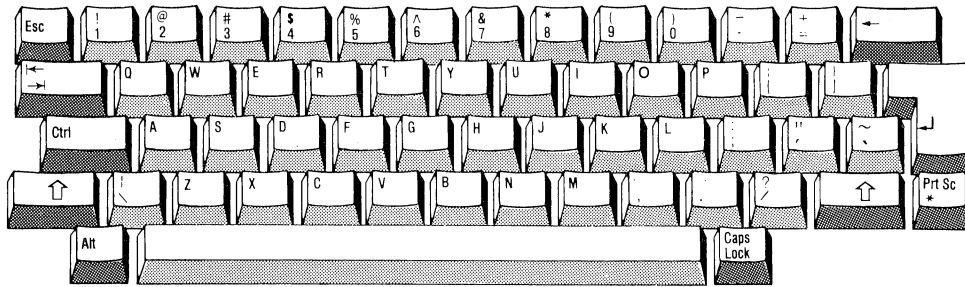
**Figure 2-1
The Keyboard**



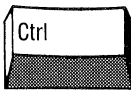
The Main Keyboard

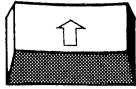
Key positions in the main keyboard are similar to a standard typewriter. Unlike a typewriter, the alphanumeric and punctuation keys are typematic, meaning if they are held down, the character is automatically repeated. GWBASIC uses some of the main keyboard keys to perform special functions. Table 2-1 lists GWBASIC's usage of some of these keys.

**Figure 2-2
Main Keyboard**



**Table 2-1
Main Keyboard Special Keys**

Key	Description
	<p>The <Ctrl> (control) key is used to enter special characters or codes not represented by keys on the keyboard. For example, <Ctrl>/<G> (<^G>) is the bell character. When this character is printed, the speaker beeps.</p> <p>The <Ctrl> key is also used with other keys to edit your programs while using the program editor (refer to the section on the GWBASIC program editor in this chapter).</p>



The main keyboard has two <Shift> keys. Pressing either of the <Shift> keys shifts the alphanumeric and punctuation keys to the upper-case mode. Alphabet characters are displayed as capital letters. All other character keys when pressed with the <Shift> key, display the character shown on the upper portion of the key. If the upper-case mode is set with the <Caps Lock> key, the <Shift> keys shift the alphabet characters to lower-case.



The <Alt> (alternate) key may be used to enter commonly-used BASIC commands or statements using a single keystroke. The BASIC keywords are entered by holding down the <Alt> key and then pressing one of the alphabet keys (A-Z or a-z). The BASIC keywords are assigned as follows:

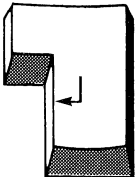
A	AUTO	N	NEXT
B	BSAVE	O	OPEN
C	COLOR	P	PRINT
D	DELETE	Q	(none)
E	ELSE	R	RUN
F	FOR	S	SCREEN
G	GOTO	T	THEN
H	HEX\$	U	USING
I	INPUT	V	VAL
J	(none)	W	WIDTH
K	KEY	X	XOR
L	LOCATE	Y	(none)
M	MOTOR	Z	(none)



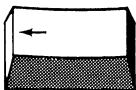
The <Caps Lock> key locks characters A through Z in the upper-case mode. Pressing <Caps Lock> again returns you to the lower-case mode. The <Caps Lock> key switches or toggles between lower- and upper-case for the characters A through Z.



Pressing the <Shift> key and <PrtSc> (print screen) key simultaneously (<Shift>/<PrtSc>) sends a copy of the information displayed on the screen to the printer (LPT1:).



The <Enter> or <CR> (carriage return) key is used to send a command to the GWBASIC interpreter, enter a line into the program in memory, or to place a newly edited program line into memory in place of the line currently there.

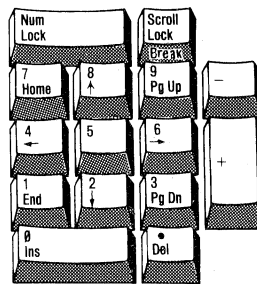


The <Backspace> key erases the character to the left of the cursor and moves the cursor one position to the left.

Numeric Keypad

The numeric keypad keys can be operated in two ways: as typical numerical keys, or as cursor movement and editing keys. You can use the <Num Lock> key to switch or toggle between the two modes. While using GWBasic, you might find it convenient to use the numeric keypad in the cursor movement and editing mode for easy program editing. The functions of the numeric keypad keys are discussed in detail in the section on the program editor later in this chapter.

**Figure 2-3
Numeric Keypad**



Function Keys

GWBasic allows the function keys, labeled F1 through F10, to be used as **soft keys** or as program interrupt keys. When GWBasic is initially started, the function keys are set as soft keys and are displayed at the bottom of the screen. Soft keys allow you to enter GWBasic keywords with a single keystroke. For more information about soft keys and the values initially assigned to them, refer to the KEY statement in Chapter 5.

The function keys can also be used as interrupt keys using the ON KEY(n) statement. In this usage, GWBasic continually checks to see if the function keys have been pressed. If a key is pressed, program execution is transferred to the program segment specified in the ON KEY statement. For a more complete explanation of how to use the function keys as interrupts, refer to the ON KEY(n) statement in Chapter 5.

Special Key Combinations

Certain key combinations perform special hardware related functions. Table 2-2 lists the key combinations and their functions.

**Table 2-2
Special Key Combinations**

Keys	Function	Description
<Ctrl>/<Alt>/	Reset	Stops all program activity and the computer loads the operating system from the diskette in drive A.
<Ctrl>/<Num Lock>	Pause	Stops the scrolling of the screen display so you can read the screen. Press any character key to continue.
<Ctrl>/<Break>	Break	Interrupts the current program and returns control to the GWBASIC command level.
<Ctrl>/<PrtSc>	Echo	Pressing the <Ctrl>/<PrtSc> toggles in and out of the Echo mode. When in the Echo mode, everything that is displayed on the screen is also sent to the printer (LPT1:).

GWBASIC PROGRAM EDITOR

Whenever you are entering text at the GWBASIC command level, you are using the program editor. The GWBASIC program editor is called a screen line editor. This means you can edit a line anywhere on the screen, but the edits are entered one line at a time.

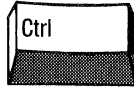


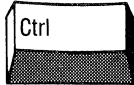



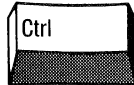

The blinking underline, called the **cursor**, indicates where the next character will be displayed or deleted on the screen. The program editor keys are used to move the cursor around the screen to edit lines. The line the cursor is on is placed in memory (or sent to the interpreter when in the direct mode) when the <Enter> key is pressed.

The program editor can be a powerful tool during program development. To become familiar with the features of the program editor, read through the following editing key descriptions and then practice using them on a sample program.

Program Editor Keys

Table 2-3 lists the editing keys used by the program editor. It describes how they are used to move the cursor around the screen to add, change, or delete characters or lines.

Table 2-3
Program Editor Keys

Key	Function
 	<p>Performs the Break function, returning you to the command level without saving changes to the line currently being edited.</p>
	<p>The <Home> key moves the cursor to the upper-left corner of the screen, called the Home position.</p>
 	<p>Clears the screen and moves the cursor to the Home position.</p>
	<p>The <Cursor Up> key moves the cursor to the character position one line above the current position.</p>
	<p>The <Cursor Left> key moves the cursor one character position to the left without deleting any characters. If the cursor is presently at the left edge of the screen, the cursor moves to the right edge of the screen on the preceding line.</p>
 	<p>The <Previous Word> key sequence moves the cursor left to the previous word. The previous word is the first letter or number to the left of the cursor that is preceded by a space or a special character. For example, consider the line</p> <pre>10 PRINT TAB(15) "HELLO" _</pre> <p>Pressing <Ctrl>/<Cursor Left> moves the cursor to the H in HELLO.</p> <pre>10 PRINT TAB(15) "HELLO"</pre> <p>Pressing <Ctrl>/<Cursor Left> again moves the cursor to the 1 in 15.</p> <pre>10 PRINT TAB(15) "HELLO"</pre>



The <Cursor Right> key moves the cursor one character position to the right without deleting any characters. If the cursor is presently at the right edge of the screen, the cursor is moved to the left edge of the screen on the next line down.



The <Next Word> key sequence moves the cursor right to the next word. The next word is the first letter or number to the right of the cursor that is preceded by a space or a special character. For example, consider the line

```
10 PRINT TAB(15) "HELLO"
```

Pressing <Ctrl>/<Cursor Right> moves the cursor to the T in TAB.

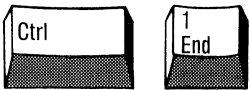
```
10 PRINT TAB(15) "HELLO"
```

Pressing the <Ctrl>/<Cursor Right> again moves the cursor to the 1 in 15.

```
10 PRINT TAB(15) "HELLO"
```



The <End> key moves the cursor one position to the right of the last character on the logical line the cursor is presently on. Characters can then be added to the end of the line.



The <Ctrl>/<End> key sequence erases all the characters from the current cursor position to the end of the logical line.



The <Cursor Down> key moves the cursor to the character position one line below the current cursor position.



The <Ins> (insert) key switches or toggles between the insert and overwrite modes. In the overwrite mode, a character typed at the keyboard replaces the character at the cursor position. The overwrite mode is the default mode for the program editor.

In the insert mode, a character typed at the keyboard is inserted into the current line at the cursor position. When a character is inserted, the cursor, the character it was on, and all the characters to the right of the cursor are moved one position to the right. If the inserted characters increase the size of the line to greater than 80 characters, the extra characters are moved to the beginning of the next line. This is known as **line wrapping**.

To make it easy to identify when you are in the insert mode, the cursor is changed from an underline to a square covering the lower half of the character box.

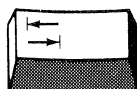
The Insert mode is automatically turned off if any of the cursor movement keys or the <Enter> key is pressed.



The (delete) key deletes the character at the current cursor position. All characters to the right of the deleted character are moved one position to the left. If the logical line spans more than one line on the screen, characters move up to fill any space left at the end of a preceding line.



The <Esc> (escape) key erases from the screen the logical line the cursor is currently on. The line is not sent to the interpreter, and if it is a program line, it is not erased from the program in memory. The cursor is returned to the beginning of the line. This provides you with a blank line so you can enter a command or new program line.



The <Tab> key moves the cursor to the next tab stop. Tab stops occur every eight character positions (1,9,17,...). In the insert mode, the <Tab> key inserts spaces from the current cursor position to the next tab stop. For example, consider the program line

```
10 REM THIS IS AN EXAMPLE
```

The cursor is presently at position 10 on the line. Pressing the <Tab> key while in the insert mode inserts seven spaces and moves the characters to the right of the I in THIS to the right.

```
10 REM TH      IS IS AN EXAMPLE
```

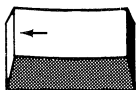
If the added spaces increase the size of the line to greater than 80 characters, line wrapping occurs as described under the <Ins> key.

In the overwrite mode, the <Tab> key moves the cursor over characters until it reaches the next tab stop. Using the above example:

```
10 REM THIS IS AN EXAMPLE
```

Pressing the <Tab> key moves the cursor to the 17th position, which is the N in AN.

```
10 REM THIS IS AN EXAMPLE
```



The <Backspace> key deletes the character to the left of the cursor and moves the cursor one position to the left. All the characters to the right of the deleted character are moved one position to the left. If the logical line spans more than one line on the screen, characters move up to fill any space at the end of the preceding line.

Entering Text Using the Program Editor

Since any text entered at the GWBASIC command level is processed by the program editor, you can use the program editor keys to make changes to the text. GWBASIC is at the command level after the Ok prompt is displayed, and remains at the command level until the RUN command is given.

GWBASIC treats an entered string of text as a **logical line**. A logical line can contain up to 255 characters. This means a logical line can extend to more than one physical line on the screen. If you enter more than 80 characters on a screen line, the cursor wraps down to the beginning of the next line and the logical line continues on that line. The logical line is not processed until a terminating <Enter> is pressed.

You can also use a line feed (<Ctrl>/<Enter>) to continue a logical line on the next screen line. Entering a line feed fills the remainder on the current line with blanks and moves the cursor to the beginning of the next line. The added blanks are included in the 255 allowed characters, but the line feed character is not added. Again, the logical line is not processed until a terminating <Enter> is pressed.

Changing Characters on the Current Line

If you are entering a line of text and discover you typed a wrong character, you can use the cursor movement keys to go back and type the correct character over the wrong character. For example, suppose you are entering the command:

```
RUN "V:PAYR_
```

A V was entered instead of a B for the drive designation. To correct it, press the previous word key sequence twice (<Ctrl>/<Cursor Left>) to move the cursor back to the V.

```
RUN "V:PAYR
```

Since the program editor is normally in the overwrite mode, simply type a B over the V to correct the mistake.

```
RUN "B:PAYR
```

Now press the <End> key to return to the end of the line to continue entering the command.

```
RUN "B:PAYR_
```

Finish typing in the command and then press the <Enter> key to process the command.

```
RUN "B:PAYROLL.BAS"
```

Deleting Characters on the Current Line

If you notice you have typed an extra character in the line you are entering, you can use the key or the <Backspace> key to delete it. For example, suppose you are entering the following command:

```
LOAD "B:PAYYROLL_
```

To delete the extra Y, press the <Cursor Left> key five times to move the cursor back to the second Y.

```
LOAD "B:PAYYROLL
```

Press the key to delete the extra Y.

```
LOAD "B:PAYROLL
```

Press the <End> key to return to the end of the line to continue entering the command. Suppose you type an extra character again:

```
LOAD "B:PAYROLL.BAA_
```

To delete the extra A, press the <Backspace> key.

```
LOAD "B:PAYROLL.BA_
```

Now continue entering the command.

```
LOAD "PAYROLL.BAS" _
```

Adding Characters to the Current Line

If you are entering text and notice that you have omitted a character, the insert mode can be used to insert characters into the middle of a line. Suppose you are entering the following command:

```
LOAD "B:PAYROL.BAS" _
```

You notice you forgot the second L in PAYROLL. Press the <Cursor Left> key five times to move the cursor back to the period after the L.

```
LOAD "B:PAYROL_ BAS"
```

Now press the <Ins> key to enter the insert mode and type in the missing L.

```
LOAD "B:PAYROLL_ BAS"
```

The command can now be processed by pressing the <Enter> key. **Note, the cursor does not have to be at the end of the line to enter the whole line for processing.**

Deleting a Portion of the Current Line

The <Ctrl>/<End> key sequence can be used to delete text from the cursor position to the end of the line. Suppose you are working on a program and decide you want to edit line 240. You can display line 240 on the screen for editing by using the EDIT command.

```
EDIT 240
240 PRINT TAB(20) "Main Editing Menu" : PRINT : PRINT
```

You have decided you only want one blank line after the menu header, so you need to delete the second PRINT statement. One way of doing this is to move the cursor to the space after the first PRINT statement using the <Cursor Right> key.

```
240 PRINT TAB(20) "Main Editing Menu" : PRINT_: PRINT
```

Now press the <Ctrl>/<End> key sequence to delete the characters from the cursor position to the end of the line.

```
240 PRINT TAB(20) "Main Editing Menu" : PRINT_
```

Now press the <Enter> key to place the new program line in memory.

Cancelling the Current Line

If you decide you do not want to process the line or command you are presently entering, press the <Esc> key. Suppose you had entered the following command:

```
DELETE 120-200_
```

To cancel the command and not delete lines 120 through 200, press the <Esc> key.

The line is deleted from the screen and the cursor returns to the beginning of the line.

ENTERING OR CHANGING A BASIC PROGRAM

Lines of text entered that begin with a line number are considered BASIC program lines and are stored in memory as part of the current program. A program line can contain a maximum of 255 characters, including the line number and the terminating <Enter>. If more than 255 characters are entered, the program editor will truncate the extra characters when the terminating <Enter> is pressed. (Note: the extra characters will still appear on the screen.)

BASIC keywords (statements, commands, and functions) and variable names are stored in memory as upper-case letters. The program editor converts all text entered in lower-case letters to upper-case letters except for remarks, items in DATA statements, and strings enclosed in quotes.

Adding a New Program Line

To add a program line to your program, enter a valid line number (0-65529), at least one space, the desired program text, and a terminating <Enter>. For example:

```
10 REM This is the first line of my program
```

When the <Enter> key is pressed, the program editor saves this line as line number 10 of your program. The program editor does not check the program line for proper syntax (correctness) before storing it in memory. Syntax is checked as the line is executed when you RUN your program.

If you enter a line with the same line number as a line already stored in memory, the new line replaces the old one.

If you try to add a line to your program when there is no more room in memory, an "Out of memory" error message is displayed and the new line is not added.

The AUTO command (described in Chapter 5) can be used to aid in program entry. The AUTO command automatically enters the line number and following space on each line for you.

Changing an Existing Program Line

Program lines can be changed in one of three ways. The first method, described above, is by entering a line with the same line number as a line already stored in memory. The new line replaces the existing line in memory. For example, if you enter

```
10 REM Program SALES FORECAST
```

the new line 10 REMark statement replaces the previous one (10 REM This is the first line of my program).

The second method is using the GWBasic EDIT command. Entering

```
EDIT 10
```

lists the current line 10 from memory on the screen and places the cursor under the 1 in 10. The program editor keys can then be used to make the desired changes. When the changes are made, press the <Enter> key to place the edited version of the program line in memory. This method also makes it easy to duplicate or move a line by changing only the line number. For example, suppose you find that you need the statement in line number 180 again in line number 430. To save typing, enter

```
EDIT 180
180 If PLUS = 0 THEN PRINT "No Positive Values"
```

To duplicate this line as line number 430, type 43 (remember, the overwrite mode is the program editor's default mode).

```
EDIT 180
430 IF PLUS = 0 THEN PRINT "No Positive Values"
```

Press the <Enter> key to enter the line as line number 430 (Note that the cursor does not have to be at the end of the line when the <Enter> key is pressed). Lines 180 and 430 now contain the same IF...THEN statement.

The third method is using the full screen editing features of the program editor. If the line is not currently displayed, the GWBASIC LIST command can be used to display the line, or a range of lines. Once the line is displayed on the screen, use the program editor cursor movement keys to move the cursor to the line requiring change. Use the editing keys to make the required changes. Press the <Enter> key to place the edited version in memory. For example, suppose you have just listed the following segment of your program:

```
LIST 110-130
110 FOR INDEX = 1 TO 10
120 IF TEST(INDEZ) < 0 THEN MINUS = MINUS + 1 ELSE
    PLUS = PLUS + 1
130 NEXT INDEX
Ok
```

You see that the reason the FOR...NEXT loop is not working properly is that you misspelled the variable INDEX in line 120. Press the <Cursor Up> key four times to move the cursor to the first screen line of logical line number 120. Press the Next Word key sequence (<Ctrl>/<Cursor Right>) three times to move to the I in INDEZ and the <Cursor Right> key four times to move to the Z in INDEZ. Now type X to replace the Z and press the <Enter> key to store the newly-corrected version in memory. (Note that the cursor does not have to be at the end of the logical line when the <Enter> key is pressed. The program editor knows where the logical line ends and processes the whole line.)

NOTE! When you edit lines using the above techniques, you only change the program in memory. To save the edited version of the program on disk, use the GWBASIC SAVE command.

Deleting a Program Line

There are two methods of deleting program lines from the program in memory. One method is to enter the line number of the line to be deleted and then press the <Enter> key. For example, entering:

```
110
```

and pressing the <Enter> key deletes line number 110 from the program in memory. If line 110 does not exist in memory, the error message "Undefined line number" would be displayed.

The second method is using the GWBASIC DELETE command. The DELETE command can be used to delete a single line, or a group of lines. Refer to Chapter 5 for instructions on how to use the DELETE command.

Syntax Errors

When the GW BASIC interpreter discovers a syntax error while running your program, the program is halted, an error message is displayed, and the line in error is listed. For example, if you misspell the BASIC keyword PRINT in line number 90 of your program, the following is displayed on your screen:

```
RUN
Syntax error in 90
Ok
90 PRIT "Your monthly payment is" PAYMENT
```

The program editor displays the line containing the error and then positions the cursor under the first digit of the line number. You can now use the <Cursor Right> key to move the cursor to the T in PRINT. Press the <Ins> key to enter the insert mode, and type N. Now press the <Enter> key to store the corrected line in memory.

C

C

C

3. GENERAL PROGRAMMING INFORMATION

This chapter covers a variety of topics you need to know to program using the GWBASIC language. Included are the program line format, the GWBASIC character set, constants and variables, and expressions and operators.

LINE FORMAT

GWBASIC program lines have the following format (square brackets indicate optional input):

```
nnnnn [BASIC statement[:BASIC statement]...] ['comment]
```

and are terminated by pressing the <Enter> key.

nnnnn Represents a line number from 0 to 65529. Every program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory. Line numbers are also used as references in branching and editing.

BASIC statement BASIC statements are either executable instructions or non-executable statements. Executable statements are instructions that tell GWBASIC what to do. Examples of executable statements are PRINT, INPUT, GOTO, and OPEN. Non-executable statements do not cause any program action, but can provide the program with needed information. Examples of non-executable statements are REM, DATA, and FIELD.

More than one BASIC statement may be included in one program line, but each must be separated from the previous one by a colon. For example:

```
50 FOR I = 1 TO 10 : PRINT I : NEXT I
```

'comment Comments may be added to the end of a program line by using the single quote mark (') to separate the comment from the rest of the line. When the program is being RUN, the interpreter ignores everything on the line after the quote mark. Comments are convenient ways of reminding yourself what each line or group of lines do in your program.

A program line may contain a maximum of 255 characters. This includes the line number, BASIC statements, comments, and one character position for the terminating <Enter>. The program line, or logical line, can extend to more than one physical line on the screen. If you enter more than 80 characters on a screen line, the cursor wraps down to the beginning of the next line and the logical line is continued. The logical line is not processed until the terminating <Enter> is pressed.

You can also use a line feed (<Ctrl>/<Enter>) to continue a logical line on the next screen line. Entering a line feed fills the remainder on the current line with blanks and moves the cursor to the beginning of the next line. The added blanks are included in the 255-allowed characters, but the line feed character is not added. Again, the logical line is not processed until a terminating <Enter> is pressed.

CHARACTER SET

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

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

The following keys act as special characters and have specific meanings in GWBASIC.

**Table 3-1
Special Character Keys**

Character	Meaning
	Blank (Space Bar)
=	Equals sign or assignment symbol
+	Plus sign or concatenation symbol
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Caret or exponentiation symbol
(Left parenthesis
)	Right parenthesis
%	Percent sign or integer type declaration character
#	Number (or pound) sign or double-precision type declaration character
\$	Dollar sign or string type declaration character
!	Exclamation point or single-precision type declaration character
[Left bracket
]	Right bracket
,	Comma or data item delimiter
'	Single quotation mark (apostrophe) or remark delimiter

Character	Meaning
"	Double-quotation mark or string delimiter
;	Semicolon
:	Colon or statement separator
&	Ampersand
?	Question mark or PRINT statement abbreviation
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At sign
_	Underscore

GWBASIC is also capable of displaying many other characters that have no special meaning in the BASIC language. Refer to Appendix B for a complete listing of the GWBASIC character set.

CONSTANTS

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

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

Examples:

```
"HELLO"
"$25,000.00"
"Number of Employees"
"B:DATAFILE.DAT"
```

Numeric constants are positive or negative numbers. A plus sign (+) is optional on positive numbers. GWBASIC numeric constants cannot contain commas. There are five types of numeric constants:

1. Integer Whole numbers between -32768 and 32767, inclusive. Integer constants do not contain decimal points.

2. Fixed-point Positive or negative real numbers; that is, numbers that contain decimal points.

3. Floating-point

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally-signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally-signed integer (the exponent). Double-precision floating point constants use the letter D instead of E. The allowable range for floating-point constants is from 2.938736E-39 to 1.701412E+38.

For example:

4.35E-2

In this example, 4.35 is the mantissa, and -2 is the exponent. The number is read as 4.35 times ten to the negative two power and can also be written as .0435.

4. Hex

Hexadecimal numbers with up to four digits and preceded by the &H prefix.

Examples:

&H76
&H32B
&HFFFF

5. Octal

Octal numbers with up to six digits and preceded by the & or &O (letter O) prefix.

Examples:

&347 or &O347
&1234 or &O1234

Precision For Numeric Constants

Numeric constants are stored as either integer, single-precision, or double-precision numbers. Constants entered as integer, hex, or octal numbers are stored in two bytes of memory and are treated as whole numbers. Single-precision numeric constants are stored in four bytes of memory with seven digits of precision and are displayed with seven digits of precision. Double-precision numeric constants are stored in eight bytes of memory with 17 digits of precision and are displayed with up to 16 digits.

A single-precision constant is any non-integer type numeric constant that is written with

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

Examples:

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

A double-precision constant is any numeric constant that is written with

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

Examples:

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

VARIABLES

Variables are names used to represent values used in a BASIC program. Variables can be used to represent both numeric and string constants. Your program may assign a value to a variable using the BASIC LET statement, using a data input statement, or as the result of a calculation.

Variable Names

GWBASIC variable names may be any length, but only the first 40 characters are significant. Variable names can contain letters, numbers, and the decimal point. The first character of a variable name must be a letter.

Special type declaration characters that identify the type of variable are also allowed as the last character of the variable name. If a type declaration character is used, the variable name cannot exceed 40 characters (with the type declaration character as the 40th character). See the following section for more information on declaring variable types.

GWBasic has certain keywords, called reserved words, that cannot be used as variable names. Reserved words include GWBasic commands, statements, function names, and operator names (see Appendix D for a listing of the GWBasic reserved words). Reserved words can be embedded within a variable name, but cannot be used alone as a variable name. If a variable begins with FN, it is assumed to be a call to a user-defined function (see the DEF FN statement in Chapter 5).

Examples:

Illegal

Legal

DATA = 5

NEW.DATA = 5

ERROR\$ = "OUT OF MEMORY"

ERROR.MES\$ = "OUT OF MEMORY"

Note, a variable name cannot be a reserved word with a type declaration character at the end.

Declaring Variable Types

Variables may represent either a numeric value or a string. The variable name determines its type.

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

A\$ = "SALES REPORT"

The dollar sign is the string variable type declaration character. It declares that the variable represents a string.

The length of a string variable is not fixed and can be from 0 to 255 characters. If you try to use a string variable before you assign a value to it, it will be assigned a null value. A null string contains no characters and has length zero.

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

- % Integer variable (2 bytes)
- ! Single-precision variable (4 bytes)
- # Double-precision variable (8 bytes)

If you use a numeric variable before before you assign a value to it, it is assigned a value of zero.

If a type declaration character is not included at the end of a variable name, GWBasic declares the variable as a single-precision numeric variable.

Examples of GWBASIC variable names:

```

PI#           Declares a double-precision value
MINIMUM!     Declares a single-precision value
LIMIT%       Declares an integer value
N$           Declares a string value
ABC          Represents a single-precision value
    
```

Variable types may also be declared by using the GWBASIC DEFINT, DEFSTR, DEFSNG, and DEFDBL statements in a program. These statements are described in detail in Chapter 5 under DEFTYPE statements.

Array Variables

An array is a group or table of values referenced by the same variable name. Each value, or **element**, in an array is itself a variable of the type indicated by the array variable name. Each element can be used in any expression, statement, or function that uses variables.

The size or dimension of an array variable is usually set using the DIM statement. The DIM statement declares the array variable type, the number of dimensions, and the number of elements in each dimension. Each element of the array is referenced by the array variable name and subscripts or indexes into the array. The subscripts are integers or expressions that evaluate to integer values. The number of subscripts indicate the number of dimensions in the array. For example:

```
DIM A$(4)
```

This DIM statement declares a one-dimensional string array with five elements. In the default option base (option base 0), the first element is referenced by the subscript 0; therefore, the five elements would be referenced as A\$(0), A\$(1), A\$(2), A\$(3), and A\$(4). The dimension statement sets aside space for the array and sets the initial values of each of the elements to zero (string array elements are set to the null value).

```
DIM B(2,2)
```

This DIM statement declares a two-dimensional numeric array with single-precision values and three elements in each dimension (assuming option base 0). The following table could be used to represent this array.

	dimension 2	0	1	2
	0	2.0	3.0	2.5
dimension 1	1	3.2	3.4	3.7
	2	1.8	2.2	1.6

The value 3.0 would be represented by the variable name B(0,1).

GWBasic allows a maximum of 255 dimensions per array, with a maximum of 32767 elements per dimension.

If an array name is encountered in a BASIC program before a DIM statement declares the array, GWBasic automatically dimensions the array with a maximum subscript value of 10. With the default zero option base, this sets an array with eleven elements per dimension. For example, if the following statements are encountered

```
115 FOR COUNTER = 1 TO 5
120 INPUT SCORES(COUNTER)
125 NEXT COUNTER
```

and the array SCORES are not declared in a DIM statement, GWBasic dimensions SCORES as a one-dimensional numeric array with eleven elements numbered SCORES(0) through SCORES(10).

NUMERIC TYPE CONVERSION

When necessary, GWBasic converts a numeric constant from one type to another. The following rules and examples apply to conversions.

1. 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. When a higher precision value is assigned to a lower precision variable, rounding occurs as opposed to truncation.

Examples:

10 A% = 23.42	10 B% = 55.88
20 PRINT A%	20 PRINT B%
RUN	RUN
23	56
Ok	Ok

2. When converting from a lower precision to a higher precision number, the resulting higher precision number cannot be more accurate than the original lower precision number. For example, if a double-precision variable is assigned a single-precision value, only the first seven digits of the converted number are valid. This is because only seven digits of accuracy are supplied with the single-precision value. The absolute value of the difference between the printed double-precision number and the original single-precision value are less than 6.3E-8 times the original single-precision value. This is stated mathematically as

$$ABS(A\# - A) < 6.3E-8 * A$$

Example:

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

3. When an expression is evaluated, all of the operands in an arithmetic or relational operation are converted to the precision of the most precise operand. The result of an arithmetic operation is returned to this degree of precision.

Examples:

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

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

```
10 D = 6# / 7
20 PRINT D
RUN
 .857143
Ok
```

The arithmetic operation was performed in double-precision, and the result was rounded to single-precision and returned to D.

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

EXPRESSIONS AND OPERATORS

An expression may be a string or numeric constant, a variable, or a combination of constants and variables with operators. An expression always produces a single value.

Operators perform mathematical or logical operations on values. GWBASIC operators may be divided into four categories:

1. Arithmetic
2. Relational

3. Logical

4. Functional

Arithmetic Operators

The arithmetic operators perform the basic arithmetic operations we are all familiar with, such as addition and subtraction. The arithmetic operators GWBASIC recognizes are listed below in order of precedence. Precedence is the order in which they are evaluated in an expression.

Operator	Operation	Sample Expression
^	Exponentiation	$10^2 = 100$
-	Negation	-2
*, /	Multiplication, Floating-point Division	$2 * 3 = 6$ $6 / 3 = 2$
\	Integer division	$3 \setminus 2 = 1$
MOD	Modulus arithmetic	$5 \text{ mod } 2 = 1$ (5/2=2 with remainder 1)
+, -	Addition, Subtraction	$3 + 2 = 5$ $3 - 2 = 1$

If two or more operations of the same level appear in an expression, they are performed in order from left to right.

You can change the order of evaluation by using parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained. For example, in the expression

$$T - 1 / Z$$

the variable Z is divided into 1 first. The result of the division is then subtracted from the variable T. If the expression is rewritten with parentheses as

$$(T - 1) / Z$$

first 1 is subtracted from variable T, then the variable Z is divided into the result of the subtraction.

The following list gives some sample algebraic expressions and their GWBASIC counterparts.

Algebraic Expression	BASIC Expression
$X + 2Y$	$X + 2 * Y$
$X - \frac{Y}{Z}$	$X - Y / Z$
$\frac{XY}{Z}$	$X * Y / Z$
$\frac{X + Y}{Z}$	$(X + Y) / Z$
$(X^2)Y$	$(X^2)^Y$
$X(-Y)$	$X * (-Y)$ Two consecutive operators must be separated by parentheses.

Integer Division And Modulus Arithmetic

In addition to the six standard operators (addition, subtraction, multiplication, division, negation, exponentiation), GWBASIC supports integer division and modulus arithmetic.

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

Examples:

```
10 FIRST = 10 \ 4
20 SECOND = 25.68 \ 6.99
30 PRINT FIRST SECOND
RUN
  2 3
OK
```

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

Examples:

```
Ok
PRINT 7 MOD 2          (7 / 2 = 3 with a remainder of 1)
  1
Ok
PRINT 25.68 MOD 6.99  (26 / 7 = 3 with a remainder of 5)
  5
Ok
```

Relational Operators

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

The relational operators are as follows:

Operator	Relation Tested	Example
=	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 LET statement in Chapter 5.)

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

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

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

More examples:

```

IF 2 > 1 THEN PRINT "CORRECT" ELSE PRINT "WRONG"
CORRECT
Ok
PRINT 2 > 1
-1 (Remember -1 indicates True)
Ok
PRINT 2 < 1
0 (Remember 0 indicates False)
Ok
IF "A"<"B" THEN PRINT "TRUE" ELSE PRINT "FALSE"
TRUE
Ok
    
```

The last example shows how string values can be compared. String characters are compared by comparing their ASCII code value (see Appendix B for a listing of the ASCII character codes). A character with a smaller ASCII code value is considered less than a character with larger ASCII code value; therefore, letters at the beginning of the alphabet are considered less than letters at the end of the alphabet. This also means that numbers in a string are less than letters and upper-case letters are less than lower-case letters.

Multi-character strings are compared character by character. When the corresponding characters of both strings have the same ASCII codes, the strings are considered equal. If corresponding characters are found with different ASCII codes, the string containing the character with the smaller code value is considered smaller, or less than the other string. When the end of one string is reached before a difference in the strings is found, the shorter string is considered less than the longer string.

```
Examples:      "3" < "C"
               "a" > "A"
               "A" < "A "
               "Filename" = "Filename"
               "DOG" > "CAT"
               "HOME" < "HOMEWARD"
               "HOMES" > "HOMEGROWN"

               PRINT 5>"F"
               Type mismatch
               Ok
```

The last example shows what happens when you try to compare a number to a string.

Logical Operators

Logical operators perform logical, or Boolean, operations on numeric values. The logical operator performs a bit-by-bit calculation (this is explained in more detail in the next section) and returns a result which is either true (non-zero) or false (zero). In an expression, logical operations are performed after arithmetic and relational operations. The result of a logical operation is determined as shown in Table 3-2. (T is used to represent true, or non-zero values. F is used to represent false, or zero values.) The operators are listed in order of precedence.

**Table 3-2
Logical Operator Truth Table**

NOT	<u>X</u>	<u>NOT X</u>
	T	F
	F	T

AND	<u>X</u>	<u>Y</u>	<u>X AND Y</u>
	T	T	T
	T	F	F
	F	T	F
	F	F	F

OR	<u>X</u>	<u>Y</u>	<u>X OR Y</u>
	T	T	T
	T	F	T
	F	T	T
	F	F	F

XOR	<u>X</u>	<u>Y</u>	<u>X XOR Y</u>
	T	T	F
	T	F	T
	F	T	T
	F	F	F

EQV	<u>X</u>	<u>Y</u>	<u>X EQV Y</u>
	T	T	T
	T	F	F
	F	T	F
	F	F	T

IMP	<u>X</u>	<u>Y</u>	<u>X IMP Y</u>
	T	T	T
	T	F	F
	F	T	T
	F	F	T

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 (see the IF statement in Chapter 5). For example:

```
140 INPUT "Enter Y for YES, N for NO",ANSWER$
145 IF (ANSWER$ = "Y") OR (ANSWER$ = "y") THEN GOSUB 450
```

In this example, if the response to the INPUT statement is Y or y, the program transfers control to line 450.

```
50 INPUT "Enter the 5 digit account number - ",ACCOUNT$
55 IF LEN(ACCOUNT$) = 5 AND VAL(ACCOUNT$) >= 10000 THEN 65
60 PRINT ACCOUNT$ "is not valid" : GOTO 50
```

Here the response to the input statement is checked for validity. The program will continue only if ACCOUNT\$ has a value greater than or equal to 10000 and is 5 digits in length.

How Logical Operators Work

Logical operators work by first converting their operands to unsigned integer values. The operands must be in the range -32768 to 32767 or an "Overflow" error message is displayed. The integer values are stored in two bytes of memory, meaning each integer value is represented by 16 binary digits (the two's complement format is used for negative values). The given logical operation is performed on these integer values by performing the logical operation on each corresponding bit; therefore, each bit of the result is determined by the corresponding bits in the two operands. In relation to the operator truth tables listed in Table 3-2, a 1 bit corresponds to a true value, and a 0 bit corresponds to a false value.

This allows logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to mask all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to merge two bytes to create a particular binary value. The following examples demonstrate how the logical operators work.

```

63 AND 16 = 16          00000000 00111111   (63)
                        AND  00000000 00010000   (16)
                        =   00000000 00010000   (16)

4 OR 2 = 6             00000000 00000100   (4)
                        OR   00000000 00000010   (2)
                        =   00000000 00000110   (6)

-1 AND 8 = 8          11111111 11111111   (-1)
                        AND  00000000 00001000   (8)
                        =   00000000 00001000   (8)
    
```

To understand how to find the binary representation of a negative number, the following equation can be used:

$$= (\text{NOT ABS}(X)) + 1$$

Therefore, to find the representation of -1, first take the absolute value of -1, which is 1. Now perform the NOT logical operation on 1, and then add 1 to that value.

```

ABS(-1) = 1 = 00000000 00000001
NOT 1 = 11111111 11111110
(NOT 1) + 1 = 11111111 11111111 = -1
    
```

If both operands of a logical operator are 0 or -1, the result is always 0 or -1.

Functional Operators

A function is a predetermined operation that is performed on an operand. GWBASIC provides many built-in functions such as SQR (square root) and SIN (sine). The GWBASIC built-in functions are described in the alphabetic listing in Chapter 5.

GWBASIC also allows you to define your own functions using the DEF FN statement. Refer to the DEF FN statement in Chapter 5.

Order of Execution

The following is a summary of the order of precedence in which the types of numeric operations are executed in an expression, and the order of precedence of operators within each type.

1. Function calls
2. Arithmetic operations
 - 1) ^
 - 2) - (negation)
 - 3) *, /
 - 4) \
 - 5) MOD
 - 6) +, -
3. Relational operations
4. Logical operations
 - 1) NOT
 - 2) AND
 - 3) OR
 - 4) XOR
 - 5) EQV
 - 6) IMP

Operations at the same level in the listing are performed in a left-to-right order. The above order can be changed by using parentheses. Operations within parentheses are performed first.

STRING OPERATORS

GWBASIC provides two categories of string operators to allow you to combine or alter string constants: concatenation and functions.

Concatenation

Concatenation means combining serially, or in the case of strings, adding one string to the end of another string. Strings are concatenated by using the plus sign (+). For example:

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

String Functions

GWBASIC provides many built-in functions that return string values. For example, the MID\$, LEFT\$, and RIGHT\$ functions return portions of a specified string. Refer to Chapter 5 for a description of the string functions available.

The GWBASIC DEF FN statement can also be used to create your own string functions. For more information on the DEF FN statement, refer to Chapter 5.



4. INPUT AND OUTPUT

This chapter contains information on input and output (I/O) in GWBASIC. Included are the following topics:

- * text and graphics display modes
- * filenames and tree-structured directories
- * disk I/O
- * special I/O features of GWBASIC

DISPLAY SCREEN I/O

The most commonly-used output device is the display screen. GWBASIC allows you to access the different text and graphics screen modes available on your TeleVideo computer. If you have a color display, or you have connected an external color monitor, GWBASIC also allows you to display text or graphics in sixteen different colors.

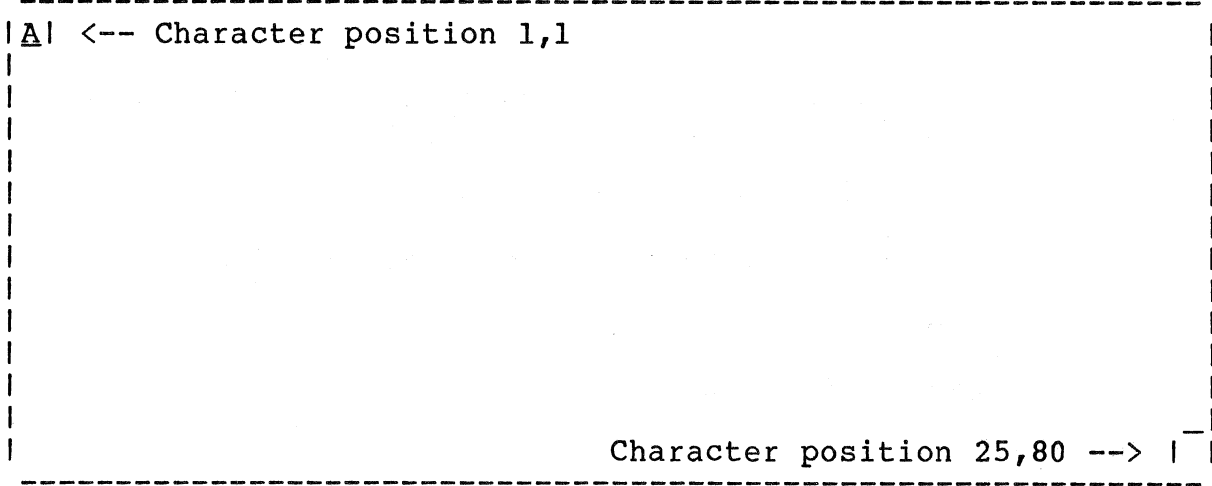
Text Modes

GWBASIC allows you to use two basic text modes; a 40-column mode, where the screen is divided into 25 vertical lines of 40 columns, or the 80-column mode, where the screen is divided into 25 vertical lines of 80 columns. When you first start GWBASIC, the screen mode is set to the 80-column text mode.

When you are using the text modes, character positions on the screen are referenced by their row (vertical line) and column (horizontal position) values. Row values are numbered 1 to 25 from top to bottom. Column values are numbered 1 to 80 (or 1 to 40) from left to right. Therefore, the character position in the upper left corner of the screen is row 1, column 1 (see Figure 4-1). These character positions are used by the LOCATE statement and the TAB, POS(0), CSRLIN, and SCREEN functions to reference positions on the screen.

Characters are normally placed on the screen using the PRINT or WRITE statements. These characters are displayed at the current cursor position, as the cursor moves from left to right across each row, and from row 1 down to row 24. When the cursor would normally move down to row 25, the screen is **scrolled** up one line instead. This means rows 1 through 24 are moved up one line, moving the characters presently in row 1 off the screen and leaving row 24 blank. The printing continues on row 24.

Figure 4-1
Text Screen



Row 25 of the screen is normally reserved for the soft key display (refer to the KEY statement in Chapter 5 for a description of soft keys). If the soft key display is turned off, you can display characters on row 25 using the LOCATE statement to move the cursor down to row 25, or you can change the screen viewing size to include row 25 using the VIEW PRINT statement.

Characters displayed on the screen in the text mode are limited to the 256 characters listed in the ASCII character code table in Appendix B. These include the regular character set consisting of letters, numbers, and punctuation, plus foreign language support characters, Greek characters, scientific characters, block graphics characters, and special game support characters.

The displayed characters are composed of two parts; a foreground and a background. The foreground is the shape of the character itself. The background is the remaining points in the character box. The text mode allows you to choose foreground and background color from 16 different colors using the COLOR statement. If you have a black and white (green) display, the different colors show up as 16 shades of gray. The colors available are:

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-intensity White

Adding sixteen to the above numbers for the foreground color produces a blinking character.

The COLOR statement can also be used to set the color of the border area of the screen. For more information on setting the text mode colors, refer to the test mode COLOR statement in Chapter 5.

The text modes also offer the use of multiple display pages. In the 40-column mode, there are 8 display pages numbered 0 to 7; in the 80-column mode, there are 4 display pages numbered 0 to 3. Multiple display pages allow you to display one page on the screen while you are writing to a different page. This allows you to instantly change the text displayed on the screen by changing the display page being viewed. For more information on the use of multiple display pages, refer to the SCREEN statement in Chapter 5.

The following is a list of the GWBasic statements that you can use to display information on the screen in the text modes:

CLS	SCREEN
COLOR	WIDTH
LOCATE	WRITE
PRINT	VIEW PRINT

The following is a list of the GWBasic functions relating to screen positions in the text modes:

CSRLIN	SPC
POS	TAB
SCREEN	

Graphics Modes

The graphics modes allow you to display the first 128 (0-127) ASCII characters and/or draw complex pictures using the GWBasic graphics statements. There are two graphics modes available; a 320 by 200 **pixel** (picture element) medium resolution mode allowing you to use four colors, or a 640 by 200 high resolution black and white mode. The graphics modes are selected using the SCREEN statement.

In addition to the display statements available in the text modes, the following graphics statements are also available:

CIRCLE	PRESET
COLOR	PSET
DRAW	PUT
GET	SCREEN
LINE	VIEW
PAINT	WINDOW

The only additional graphics function is the POINT function.

Medium Resolution

Medium resolution divides the screen into 320 pixels horizontally by 200 pixels vertically. Each pixel can be displayed in one of four colors. The four colors available to the screen are chosen using the COLOR statement. A pixel is displayed in one of the four colors by using one of the graphics statements with a color argument of 0, 1, 2, or 3. The 0 argument indicates the background color, while colors 1, 2, and 3 are selected from one of two preset palettes of three colors. Changing the background color changes every pixel displayed in color 0, while changing the selected palette changes every pixel displayed in colors 1, 2, or 3. For more information on selecting screen colors, refer to the COLOR statement in Chapter 5.

When text is displayed in medium resolution, characters are displayed in the font and format used in the 40-column text mode. Color 3 is used as the foreground color, color 0 is used as the background color.

High Resolution

High resolution divides the screen into 640 pixels horizontally by 200 pixels vertically. Each pixel can be displayed in either black, color 0, or white, color 1.

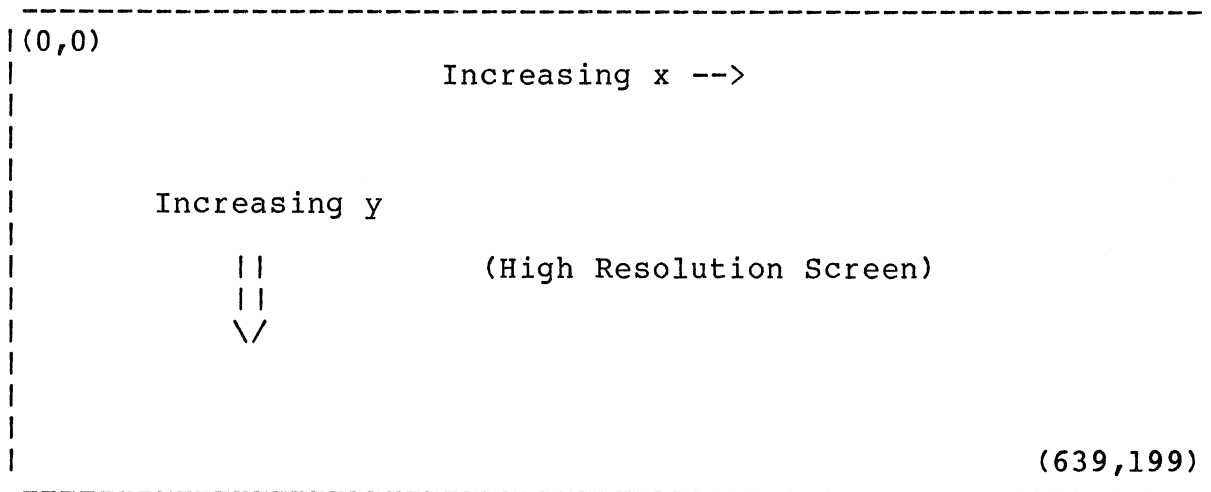
When text is displayed in high resolution, characters are displayed in the font and format used in the 80-column text mode. White is used as the foreground color and black is used as the background color; therefore, white characters are displayed on a black background.

Specifying Graphics Coordinates

The graphics statements require you specify where on the screen you want to draw. You enter this information in the form of graphic coordinates. Graphic coordinates are generally given in the form (x,y), where x is the horizontal position on the screen (0-319 in medium resolution, 0-639 in high resolution), and y is the vertical position (0-199 in both resolutions). This form of coordinates is known as the **absolute** form, because it specifies a specific point on the screen.

Positions are numbered from left to right and from top to bottom; therefore, the upper left position on the screen is (0,0) and the lower right position on the screen is (319,199) or (639,199), depending on which resolution you are in.

Figure 4-2
Graphics Screen



Note, in the text mode, character positions are referenced by (vertical position, horizontal position), whereas in the graphics mode, points are referenced by (horizontal position, vertical position).

Many of the graphics statements also allow the use of a **relative form** of coordinates. This form of coordinates gives the graphics statement an offset from the **current graphics position** and is entered in the following format:

```
STEP (xoffset,yoffset)
```

The current graphics position is the **last point referenced** by the last graphics statement used. The last point referenced is usually the last point drawn by the graphics statement (refer to each graphics statement to find out what point is considered the last point referenced for that statement). The POINT function can be used to determine the coordinates of the current graphics position. When a program is RUN from a graphics mode, or a SCREEN mode change is made to a graphics mode, the current graphics position is set to the middle of the screen; position (160,100) in medium resolution, position (320,100) in high resolution.

The xoffset and yoffset values indicate the number of pixels (points) in the x and y directions to move. Positive values of x and y move you to the right and down respectively; negative values of x and y move you to the left and up respectively. For example, the statements:

```
10 SCREEN 1
20 PSET (320,100)
30 PSET STEP(-20,0)
```


plot two points on the screen. Line 20 plots a point at position (320,100). Line 30 plots a point at position (300,100).

FILES

A file is a collection of information which is normally stored somewhere other than in the random access memory of your computer. For example, you normally store program and data files on a diskette or hard disk. Before the file can be used for input and output (I/O), you must tell GWBASIC where the information is located. Filenames and file numbers are used for this purpose.

GWBASIC supports the concept of general device I/O files. This means that any type of I/O may be treated like I/O to a file; the keyboard can be used as an input file, the printer can be used as an output file, or the serial communications port can be used as an I/O file.

Naming Files

Files are located or referenced by a file specification, or **filespec** for short. A filespec is a string expression of the form:

```
[<device>:]<filename>[.<ext>]
```

where the device tells GWBASIC where to look for the file, and the filename and extension (ext) tell GWBASIC which file to look for on the device. The device portion of the filespec is optional; if omitted, GWBASIC assumes the file is a disk file located on the TeleDOS default drive, if included, the colon must be included between the device and the filename.

When you enter a filespec as a string constant, you must enclose it in quotes. For example:

```
70 OPEN "B:SALES.DAT" FOR INPUT AS #1
```

Device Names

The device name consists of up to four characters followed by a colon (:). Table 4-1 is a list of the device names recognized by GWBASIC, and includes the device it refers to and the type of I/O available to the device.

Table 4-1
Device Names

Name	I/O	Device
KYBD:	I	Keyboard
SCRN:	O	Display screen
LPT1:	O	Parallel printer port (PARALLEL PRINTER)
LPT2:	O	A second, optional printer
LPT3:	O	A third, optional printer
COM1:	I/O	RS-232C serial port (RS-232C)
COM2:	I/O	An optional asynchronous communications adapter
A:	I/O	First diskette drive
B:	I/O	Second diskette drive
C:	I/O	First hard disk drive
D:	I/O	Second hard disk drive

Optional printers, communications adapters, and hard disk drives are attached using the expansion slot.

Filenames

Filenames used in GWBASIC conform to the naming conventions used in the TeleDOS operating system. This consists of a filename and an optional filename extension in the format:

<filename>[.<ext>]

where the extension, when included, must be separated from the filename by a period (.). The filename itself can be from one to eight characters in length, and the extension can be up to three characters in length.

If an entered extension is longer than three characters, the extra characters are truncated. If a filename is entered with more than eight characters and an extension is not included, GWBASIC inserts a period after the eighth character and uses up to the next three characters as an extension. If a filename is longer than eight characters and an extension is included, GWBASIC uses the first eight characters in the entered filename, truncates the extra letters up to the period, and then adds the entered extension.

The following characters are allowed in a filename and extension:

Letters A - Z	! @ # \$ % ^ & ()
Numbers 0 - 9	- _ ` ~ ' / { }

Some examples of valid filenames are:

PROGRAM1.BAS

2ND_TRY

(!@\$%).123

The following examples show how GWBasic truncates filenames and extensions when they are too long:

TEST_PROGRAM	becomes	TEST_PRO.GRA
TEST.PROGRAM	becomes	TEST.PRO
TEST_PROGRAM.BAS	becomes	TEST_PRO.BAS

Tree-Structured Directories

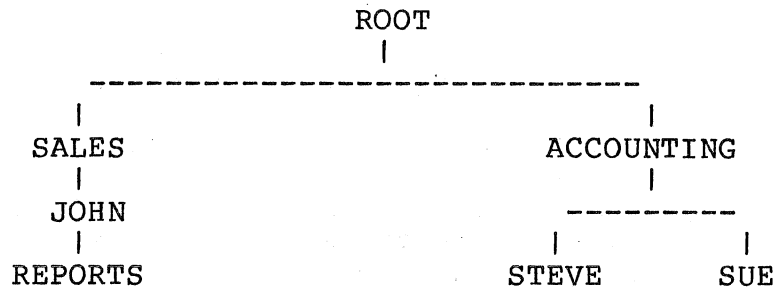
GWBasic allows you to use the tree-structured directory system of the TeleDOS operating system (see Chapter 3 of the TeleDOS User's Manual for a complete description of tree-structured directories). When using tree-structured directories, you might need to add a directory path to the filespec to tell GWBasic how to find the specified file. The filespec would then be in the following format:

[<device>:][\][<directory>[\<directory>]...[\]<filename>[.<ext>]

where

[<device>:][\][<directory>[\<directory>]...

is called a path. The path tells GWBasic what part of the directory system the file is located in. The total length of the filespec cannot exceed 128 characters. The following example shows a simple directory structure:



When a disk is formatted, a single directory is created. This directory is known as the root directory. In addition to containing the names of files, the root directory can also contain the names of other directories. These directories are called subdirectories of the root directory. A subdirectory is a file that can contain files or other subdirectories. The format for a directory name is the same as for a filename.

In the above example, SALES and ACCOUNTING are subdirectories of the root directory. STEVE and SUE are subdirectories of directory ACCOUNTING, and JOHN is a subdirectory of directory SALES. REPORTS is a subdirectory of directory JOHN.

When GWBASIC is started, the current directory is set to the current TeleDOS directory. This directory is called your current directory. To specify a file that is in a different directory, you must give the directory path from the current directory to the directory containing the file. To access a file called WEEK2.RPT in the REPORTS directory while in the root directory, the following filespec is entered:

```
"SALES\JOHN\REPORTS\WEEK2.RPT"
```

If you plan on working only on files in the REPORTS directory, you can use the CHDIR command to make the REPORTS directory the current directory.

```
CHDIR "SALES\JOHN\REPORTS"
```

GWBASIC now searches the REPORTS directory for any filespec entered without a directory path. To specify the WEEK2.RPT file, the following filespec is entered:

```
"WEEK2.RPT"
```

To specify a file SALES.RPT in the SALES directory while the REPORTS directory is the current directory, you again must indicate the directory path.

```
"\SALES\SALES.RPT"
```

In this example, a backslash is used before the SALES directory entry to indicate the directory search starts at the root directory. If the first backslash had not been entered, GWBASIC would try to find a file named SALES.RPT in a subdirectory SALES in the current directory REPORTS.

Commands for Program Files

The most common usage of filespec entries is in the commands referring to your program files. The following list gives you a brief description of their usage. For a more detailed description of these commands, refer to Chapter 5.

Table 4-2
Program File Commands

Command	Description
SAVE <filespec>	Writes the program currently in memory to the specified disk.
LOAD <filespec>	Loads the specified program file into memory from the specified disk.
RUN <filespec>	Loads the specified program from disk into memory and runs it.
MERGE <filespec>	Loads the specified program from disk into memory and merges it with the program currently in memory.
KILL <filespec>	Deletes the specified program from disk.
NAME <filespec> AS <filename>	Changes the name of the specified file.

Protected Files

GWBASIC provides a feature to let you protect your program files. When the ,P (protect) option is used when you SAVE a file, the file is saved in an encoded binary format. This format prevents the file from being listed, edited, or saved. Since there is no way of unprotecting the file, it is a good idea to save an unprotected version of the program in case you need to make changes.

File Numbers

GWBASIC performs I/O operations by referring to a file by a file number. The file number is assigned to a file through the use of the OPEN or OPEN COM statement. The file number is a unique number that is associated with the actual file while it is open for I/O. The file number is a numeric constant, a variable, or a numeric expression in the range 1 to n, where n is the maximum number of files allowed. The default value of n is 5, but n can be set to a maximum of 15 by changing the maximum number of files supported by TeleDOS. For more information on how to assign a file number to a file, refer to the OPEN and OPEN COM statements.

DISK DATA FILES

GWBasic supports two types of data files; sequential files and random access files.

Sequential Files

A sequential data file is as the name implies, a file where the data is stored sequentially, one item after the next. Sequential files are easier to learn to use, but they lack the convenience and speed when accessing data that is available with random access files.

To read a data item from a sequential file, you must start at the beginning of the file and read each item in the order it was placed in the file until you come to the item you are looking for.

The following statements and functions are used with sequential files:

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

A sequential file is created using the following steps:

- 1) Open a sequential file for output using the OPEN statement.
- 2) Place data into the file using the PRINT #, PRINT # USING, or WRITE # statements.
- 3) Close the file using the CLOSE statement. This statement makes sure any data remaining in the file buffer is written out to the file.

To access the data in a sequential file, use the following steps:

- 1) Open the file for input using the OPEN statement.
- 2) Read data from the sequential file using the INPUT # or LINE INPUT # statements.

The following simple program example shows how to create a sequential file containing a list of names.

```

10 OPEN "NAMES" FOR OUTPUT AS #1
20 PRINT "ENTER NAMES, PRESS ENTER WITHOUT AN ENTRY TO END"
30 INPUT "NAME";NAMES$
40 WHILE NAMES$ <> ""
50 WRITE #1,NAMES$
60 INPUT "NAME";NAMES$
70 WEND
80 CLOSE
RUN
ENTER NAMES, PRESS ENTER WITHOUT AN ENTRY TO END
NAME? SANDY
NAME? LYNN
NAME? SUSAN
NAME? ANN
NAME? TOM
NAME?
Ok
    
```

This program has opened a sequential file called NAMES, and placed the names SANDY, LYNN, SUSAN, ANN, and TOM in the file in that order. To read the names back out of the file, the following program could be used.

```

10 OPEN "NAMES" FOR INPUT AS #1
20 WHILE NOT EOF(1)
30 INPUT #1,NAMES$
40 PRINT NAMES$
50 WEND
60 CLOSE
RUN
SANDY
LYNN
SUSAN
ANN
TOM
Ok
    
```

This program reads the names out of the sequential file and displays them on the screen. The EOF function is used to make sure the program does not try to read past the end of the file. When a file is created, the ASCII code 26 (hex 1A) is written to the file after the last data item to mark the end of the file. The EOF function looks for this end-of-file character. If the end-of-file character is found, EOF returns a true value, otherwise EOF returns a false value. In our example, as long as EOF returns a false value, line 30 in the WHILE...WEND loop reads the next item in the file.

Changing a Sequential File

If you have a created a sequential file on disk, and later want to add more data to the file (after the file has been closed), you cannot simply open the file for output and start writing data to the file. Each time you open a sequential file for output, the file pointer points to the beginning of the file. The first data item written to the file becomes the first item in the file. Any data that was in the file is lost.

To add data items to the end of the file, you must open the file for APPEND. This opens the file for output, but sets the file pointer to the end of the file. The next data item written to the file is added on to the end of the file.

To make changes to data items currently in a sequential file, you must open the file for input, read in and store in memory every data item in the file, make the changes to the appropriate data item(s), open the file for output, and then write the entire file back out to disk.

Random Access Files

Random access files require more programming steps to use, but they provide the convenience of being able to access and change any data item in the file without having to read through all the information placed before it in the file.

Data in random access files are stored and accessed in units of storage called a record. When you place a data item in a random access file, you specify which record in the file to store the data in. To access a data item, you simply specify which record the data item is in.

Records can range in length from 1 to 32767 bytes long, depending how much information you want to put in each record. The record length is set when the file is opened with the OPEN statement.

The following statements and functions are used with random access files:

- | | |
|-------|-------|
| CLOSE | LSET |
| CVD | MKD\$ |
| CVI | MKI\$ |
| CVS | MKS\$ |
| FIELD | OPEN |
| GET | PUT |
| LOC | RSET |
| LOF | |

A random access file is created using the following steps:

- 1) Open a random access file using the OPEN statement. If a record length is not specified in the OPEN statement, a default value of 128 bytes is used.

- 2) Use the FIELD statement to allocate space in the random access file buffer. The FIELD statement assigns variable names to bytes of storage in the record; therefore, the total amount of space allocated with the FIELD statement cannot exceed the length of the record specified in the OPEN statement. The fielded variables are used to move data in and out of the file buffer.
- 3) Use LSET or RSET to place data into the file buffer. Numeric data must first be converted to a string representation using the MKI\$, MKS\$, or MKD\$ functions. MKI\$ converts an integer value to a 2-byte string, MKS\$ converts a single-precision value to a 4-byte string, and MKD\$ converts a double-precision value to an 8-byte string.
- 4) Write the data from the file buffer to the file using the PUT statement.
- 5) Close the file with the CLOSE statement to make sure all the information in the file buffer is written to file.

The following example shows how to create a random access file containing a name, age, and social security number in each record.

```

10 OPEN "INFORM.DAT" AS #1 LEN = 38      'STEP 1
20 FIELD #1,25 AS INFORM1$,2 AS INFORM2$,11 AS INFORM3$
                                           'STEP 2
30 PRINT "ENTER NAME, PRESS ENTER WITHOUT AN ENTRY TO END"
40 COUNTER% = 1
50 INPUT "NAME";NAMES$
60 WHILE NAMES$ <> ""
70 INPUT "AGE";AGE%
80 INPUT "SOCIAL SECURITY # (XXX-XX-XXXX)";SOCSEC$
90 LSET INFORM1$ = NAMES$                'STEP 3
100 LSET INFORM2$ = MKI$(AGE%)
110 LSET INFORM3$ = SOCSEC$
120 PUT #1,COUNTER%                     'STEP 4
130 COUNTER% = COUNTER% + 1
140 INPUT "NAME";NAMES$
150 WEND
160 CLOSE                                'STEP 5
    
```

In this example, each record contains a string value name, an integer value age, and a string value social security number. The variable COUNTER% is used to place each record of data into the next available record in the file.

Data in a random access file is accessed using the following steps:

- 1) Open the file for random access using the OPEN statement.

- 2) Use the FIELD statement to allocate space in the file buffer for the variables used to read the data from the file.

NOTE! In a program that performs both input and output on the same random access file, only one OPEN and one FIELD statement are required.

- 3) Use the GET statement to move the data from the file into the file buffer.
- 4) Use the variables defined in the FIELD statement to access the data in the file buffer. Numeric values must be converted back to numbers using the CVI, CVS, or CVD functions. CVI converts 2-byte strings back to integers, CVS converts 4-byte strings back to single-precision numbers, and CVD converts 8-byte strings back to double-precision numbers.

The following program lines could be used to access the data in the INFORM.DAT random access file created above.

```

10 OPEN "INFORM.DAT" AS #1 LEN = 38      'STEP 1
20 FIELD #1,25 AS NAMES$,2 AS AGE$,11 AS SOCSEC$
                                           'STEP 2
30 INPUT "ENTER THE NUMBER OF RECORDS TO READ";RECORDS%
40 PRINT "NAME" TAB(30) "AGE" TAB(40) "SOCIAL SECURITY #"
50 PRINT
60 FOR COUNTER% = 1 TO RECORDS%
70 GET #1,COUNTER%                        'STEP 3
80 PRINT NAMES$ TAB(30) CVI(AGE$) TAB(40) SOCSEC$
                                           'STEP 4
90 NEXT COUNTER%
100 CLOSE

```

The following program uses a random access file to store the assignment of company phone extensions. The phone extensions range from 101 to 299. The extension number minus 100 is used as the record number for the information on each extension. The first byte of each record contains the letter Y or N, to indicate whether that extension is being used. The next 25 bytes of the record hold the name the extension is assigned to, and the last 10 bytes contain the department the person belongs to.

```

10 'PROGRAM PHONE EXTENSIONS
20 OPEN "EXTENS" AS #1 LEN = 36
30 FIELD #1, 1 AS USED$ , 25 AS NAMES$ , 10 AS DEPT$
40 SCREEN 0 : WIDTH 80
50 CLS
60 PRINT : PRINT TAB(30) "PHONE EXTENSION MENU" : PRINT
70 PRINT TAB(30) "1 - LIST EXTENSION"

```

```
80 PRINT TAB(30) "2 - ENTER LISTING"
90 PRINT TAB(30) "3 - DELETE LISTING"
100 PRINT TAB(30) "4 - INITIALIZE FILE"
110 PRINT TAB(30) "5 - EXIT PROGRAM" : PRINT
120 PRINT TAB(30) : INPUT "ENTER SELECTION";SEL$
130 SEL = VAL(SEL$)
140 IF SEL < 1 OR SEL > 5 THEN GOTO 170
150 ON SEL GOSUB 200, 300,450,550,800
160 GOTO 50
170 PRINT : PRINT CHR$(7) TAB(30) "INVALID SELECTION"
180 LOCATE 10,1 : PRINT SPC(79) : LOCATE 10,1 : GOTO 120
200 '*** LIST EXTENSION ***
210 CLS : GOSUB 700 'GET EXTENSION NUMBER
220 GET #1,EXT - 100
230 IF USED$ = "N" THEN PRINT EXT "IS NOT BEING USED"
      : GOTO 270
240 PRINT "EXTENSION -" EXT
250 PRINT "NAME: " NAMES$
260 PRINT "DEPARTMENT: " DEPT$
270 PRINT : INPUT "PRESS ENTER TO RETURN TO THE MENU",A$
280 RETURN
300 '*** ENTER LISTING ***
310 CLS : GOSUB 700 'GET EXTENSION NUMBER
320 GET #1,EXT - 100
330 IF USED$ = "N" THEN 370
340 PRINT : PRINT EXT "IS ALREADY ASSIGNED"
350 INPUT "DO YOU WANT TO CHANGE THE ASSIGNMENT (Y/N)";ANS$
360 IF ANS$ = "N" THEN RETURN
370 INPUT "ENTER NAME";N$
380 INPUT "ENTER DEPARTMENT";D$
390 LSET USED$ = "Y"
400 LSET NAMES$ = N$
410 LSET DEPT$ = D$
420 PUT #1,EXT - 100
430 RETURN
450 '*** DELETE LISTING ***
460 CLS : GOSUB 700 'GET EXTENSION NUMBER
470 LSET USED$ = "N"
480 LSET NAMES$ = ""
490 LSET DEPT$ = ""
500 PUT #1,EXT - 100
510 RETURN
550 '*** INITIALIZE FILE ***
560 CLS : PRINT "INITIALIZING THE FILE DESTROYS ALL DATA ";
570 PRINT "IN THE FILE" : PRINT
580 INPUT "ARE YOU SURE (Y/N)";ANS$
590 IF ANS$ = "N" THEN RETURN
600 FOR I = 1 TO 199
610 LSET USED$ = "N"
620 LSET NAMES$ = ""
630 LSET DEPT$ = ""
640 PUT #1,I
650 NEXT I
660 RETURN
700 '*** ENTER EXTENSION NUMBER SUBROUTINE ***
```

```

710 INPUT "ENTER EXTENSION NUMBER (101-299)";EXT$
720 PRINT : EXT = VAL(EXT$)
730 IF EXT >= 101 AND EXT <= 299 THEN RETURN
740 PRINT : PRINT "INVALID EXTENSION NUMBER"
750 GOTO 710
800 '*** EXIT PROGRAM ***
810 RETURN 820
820 CLOSE
830 CLS : END
    
```

SPECIAL I/O FEATURES

In addition the standard input and output mentioned above, GWBasic supports I/O to the devices listed in Table 4-3.

**Table 4-3
Additional I/O Support Devices**

Device	Function/ Statement	Purpose
System Clock	DATE\$	To set or retrieve the date known to the system.
	ON TIMER	Provides event trapping based on a defined period of time having elapsed.
	TIME\$	To set or retrieve the time known to the system.
	TIMER	Returns the number of seconds that have elapsed since midnight or the last system reset.
Speaker	BEEP	Beeps the speaker.
	ON PLAY	Provides music event trapping to allow continuous background music during program execution.
	PLAY	Plays music by specifying the music to be played using a music macro language.
	SOUND	Makes the sound specified by the entered frequency and duration.
Light Pen	ON PEN	To provide event trapping for light pen activity.
	PEN	To read the position of the light pen.

Joysticks	ON STRIG	To provide event trapping for joystick activity.
	STICK	Returns the coordinates of the joysticks.
	STRIG	Returns the status of the joystick buttons.

5. BASIC COMMANDS, STATEMENTS, AND FUNCTIONS

This chapter contains descriptions of all the GWBasic commands, statements, and functions. The distinction between commands, statements, and functions will be defined as follows:

Command An instruction that returns control to the command level after the instruction has been performed. Commands are normally entered in the direct mode and generally operate on a program. For example, RUN, LIST and SAVE are commands.

Statement An instruction that is normally entered in the indirect mode as part of a program to direct the flow of the program. For example, PRINT, LET and GOTO are statements.

Function A function converts a value into some other value according to a fixed formula. The functions described in this chapter are built-in, or "intrinsic" to GWBasic. These functions may be called from any program without further definition.

The descriptions of the commands, statements, and functions are presented using the following format:

NAME

Type (Command, Statement, or Function)

Purpose Describes what the command, statement, or function does.

Format Shows the correct syntax. Syntax notation rules to remember are:

- * Items listed in upper-case letters are keywords and must be entered as listed. They may be entered in lower- or upper-case letters, or a combination of both. The program editor converts these keywords to upper-case.
- * You are to supply items listed in lower-case letters within the < > symbols.
- * Items listed in square brackets ([]) are optional.

- * Braces ({}) indicate you have a choice between two or more items. The choices will be separated by a vertical bar (|). At least one of the entries must be selected unless the entries are also enclosed in square brackets.
- * An ellipse (...) indicates the preceding item may be repeated as many times as desired.
- * All punctuation (except those items noted above) must be included as shown.
- * When the term **filespec** is used as an entry item, it refers to a combination of an optional device name, an optional path, a filename, and an optional file extension. GWBasic uses the filename conventions described in Chapter 3 of your TeleDOS User's Manual.

Comments A complete and detailed description of how the command, statement, or function is used.

Example Sample programs, program segments, or direct mode statements that demonstrate the use of the command, statement, or function.

Notes Describes special cases or provides additional pertinent information.

Many of the parameters used in the formats of the commands, statements, and functions will use the following abbreviations:

- x, y, z represent numeric expressions
- i, j, k, m, n represent integer expressions
- x\$, y\$ represent string expressions

If a single- or double-precision value is entered where an integer value is required, GWBasic rounds the fractional portion and uses the resulting integer.

**ABS
Function**

Purpose To return the absolute value of an expression.

Format ABS(x)

 x is any numeric expression.

Comments The ABS function returns the magnitude of the numeric
 expression x; therefore, the value is always positive
 or zero.

Example 10 A = -5
 20 B = ABS(7*A)
 30 PRINT B
 RUN
 35
 Ok

 In this example, the ABS function returns the absolute
 value of -35, which is 35.

**ASC
Function**

Purpose To return a numerical value that is the ASCII code for the first character in the specified string. (See Appendix B for ASCII codes.)

Format ASC(x\$)

 x\$ is any string expression.

Comments If x\$ is a null string (contains no characters), an "Illegal function call" error message is displayed.

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

 In this example, the ASC function returns the ASCII code of upper-case T, which is 84.

Notes The CHR\$ function is the complement of the ASC function and converts an ASCII code to a character.

**ATN
Function**

Purpose To return in radians the trigonometric arctangent of the specified numeric expression.

Format ATN(x)

x is any numeric expression.

Comments The evaluation of the ATN function is performed in single-precision. If the /D parameter was entered on the GWBasic command line, evaluation is performed in double-precision.

To convert radians into degrees, multiply radians by $180 / \pi$, where $\pi = 3.141593$.

Example

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

**AUTO
Command**

Purpose To automatically generate line numbers during program entry.

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

 line number is the line number AUTO begins numbering at. A period (.) may be entered to indicate the current line number.

 increment is the value AUTO uses to add to the present line number to get the next line number in the sequence.

Comments When the AUTO command is entered, AUTO displays the first line number in the sequence and a following space, and then waits for you to enter a program line. When you have entered your program line and pressed the <Enter> key, AUTO displays the next line number and the following space. The next line number is equal to the last line number plus the increment.

Each time the <Enter> key is pressed, AUTO generates the next line number in the sequence. If <Enter> is pressed without entering any text, a program line is not stored in memory for that line number.

When both <line number> and <increment> are omitted, both values default to ten (10). If the <increment> and comma are omitted, the increment defaults to ten (10). When the <line number> and comma are entered, but the <increment> is omitted, the increment defaults to the last increment used in an AUTO command. If no previous increment has been specified, the increment defaults to ten (10). When a comma and <increment> are entered, but the <line number> is omitted, the first line number defaults to zero (0).

If AUTO generates a line number that already exists in the program, an asterisk (*) is displayed after the line number to warn you that any input will replace the existing line. Pressing the <Enter> key immediately after the asterisk saves the existing line and generates the next line number.

To exit the AUTO mode and return to the command level, press the <Ctrl>/<Break> key sequence. The line in which the <Ctrl>/<Break> is entered is not saved.

**AUTO
Command**

Examples **AUTO** Generates line numbers 10, 20, 30, ...

AUTO 10,5 Generates line numbers 10, 15, 20, ...

AUTO ,2 Generates line numbers 0, 2, 4, ...

Notes When in the AUTO mode, if you move the cursor to another line number on the screen and press the <Enter> key, that line is stored in memory and AUTO resumes line numbering from that line number.

**BEEP
Statement**

Purpose To beep the speaker at 800 Hz for 1/4 second.

Format BEEP

Comments The BEEP statement sounds the ASCII bell character.
 This statement has the same effect as PRINT CHR\$(7).

Example 120 'IF X IS OUT OF RANGE, BEEP THE SPEAKER
 130 IF X < 20 THEN BEEP

 In this program segment, the program beeps the speaker
 if the value of variable X is less than 20.

**BLOAD
Command**

Purpose To load a specified memory image file into memory from disk.

Format BLOAD <filespec> [,<offset>]

filespec is a string expression for the name of the file to be loaded. The <filespec> must conform to the rules for naming files or a "Bad file name" or "File not found" error message is displayed.

offset is a numeric expression returning an unsigned integer in the range of 0 to 65535. This is an offset into the segment declared by the last DEF SEG statement indicating the address where loading will begin. If a DEF SEG has not been issued, the GWBasic data segment is used as the default value.

Comments The BLOAD statement allows a program or data that has been saved as a memory image file to be loaded anywhere in memory. A memory image file is a byte-for-byte copy of what was originally in memory. See the BSAVE Statement for information about saving memory image files.

If the <offset> is omitted, the segment address and offset contained in the file (the address specified by the BSAVE statement when the file was created) are used. Therefore, the file is loaded into the same location from which it was saved.

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

This example sets the segment address at Hex 6000 and then loads PROG1 at an offset of Hex F000 into the segment.

Notes BLOAD does not perform an address range check. It is therefore possible to load a file anywhere in memory. You must be careful not to BLOAD the file over GWBasic or TeleDOS.

BSAVE
Command

 Purpose To save the contents of the specified area of memory
 as a disk file.

Format BSAVE <filespec>,<offset>,<length>

filespec is a string expression for the name of the
 file to be saved. The <filespec> must
 conform to the rules for naming files or a
 "Bad file name" or "Too many files" error
 message is displayed and the save is aborted.

offset is a numeric expression returning an unsigned
 integer in the range of 0 to 65535. This is
 an offset into the segment declared by the
 last DEF SEG statement, indicating the
 starting address of the save. If a DEF SEG
 has not been issued, the GWBASIC data
 segment is used as the default value.

length is a numeric expression returning an unsigned
 integer in the range of 1 to 65535
 representing the length in bytes of the
 memory image to be saved.

Comments The BSAVE statement allows data or programs to be saved
 as memory image files on disk. A memory image file is
 a byte-for-byte copy of what is in memory.

If the <offset> or the <length> is omitted, a "Syntax
 error" error message is displayed and the save is
 aborted.

Example 10 'Save 256 bytes at 6000:F000 as file PROG1
 20 DEF SEG = &H6000
 30 BSAVE "PROG1",&HF000,256

This example saves 256 bytes of memory starting at
 address 6000:F000 in the file PROG1 on the default
 drive.

**CALL
Statement**

Purpose To call an assembly language subroutine.

Format CALL <variable>[(<argument>[, <argument>]...)]

variable is a numeric variable containing the starting memory address of the subroutine being called. The address is an offset into the segment defined by the last DEF SEG statement. <variable> cannot be an array variable name.

argument is the name of a variable being passed as an argument to the external subroutine.

Comments The CALL statement is one way to transfer program flow to an external subroutine. The other method is using the USR function.

Example 100 DEF SEG = &H1800
110 START = 0
120 CALL START(FIRST, LAST)

In this program segment, line 100 sets the current segment to Hex 18000. The variable START is assigned the value zero and used as an offset into the segment to indicate the starting address of the subroutine. The variables FIRST and LAST are passed as arguments to the subroutine.

**CDBL
Function**

Purpose To convert a numerical expression to a double-precision number.

Format CDBL(x)

 x is any numeric expression.

Comments The CDBL function converts the result of numeric expression x to a double-precision number using the rules for numeric type conversion as described in Chapter 3.

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

In this example, the value of CDBL(A) is only accurate to two digits to the right of the decimal point because that is the accuracy that was supplied to variable A. The additional digits are meaningless.

**CHAIN
Statement**

Purpose To call and transfer control to another program and optionally pass variables to it from the current program.

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

filespec is a string expression for the name of the program to be called. <filespec> must conform to the rules for naming files or a "Bad file name" or "Path not found" error message is displayed and the chain is aborted.

line number is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. The <line number> value is not affected by the RENUM command.

range is the first and last line numbers of the range of lines to be deleted by the DELETE option. If the last line number specified in the range does not exist, an "Illegal function call" error message is displayed. The <range> line numbers are affected by the RENUM command.

Comments With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to pass variables to the called program.

If the ALL option is used and <line number> is omitted, a comma must hold the place of <line number>. For example:

```
135 CHAIN "NEXTPROG",,ALL
```

is correct;

```
135 CHAIN "NEXTPROG",ALL
```

is incorrect. In the latter case, ALL would be considered a variable name and evaluated as a line number expression.

**CHAIN
Statement**

The MERGE option allows a subroutine to be brought into the BASIC program as an overlay. That is, the current program and the called program are merged (see the MERGE command for more information on merging programs). To transfer control to the merged subroutine, the beginning subroutine line number must be included in the CHAIN statement or control is transferred to the beginning of the resulting merged program. The called program must have been stored in an ASCII format using the /A option with the SAVE command if it is to be merged. For example:

```
185 CHAIN MERGE "OVERLAY1",500
```

would merge program OVERLAY1 from the default drive into the current program and transfer control to line number 500.

When using several overlays, it is usually desirable to delete the old overlay before the new overlay is brought in. To do this, use the DELETE option. The delete option deletes the lines in the specified range before bringing in the new program. For example:

```
185 CHAIN MERGE "OVERLAY1",500,DELETE 500-780
```

would delete lines 500 through 780 from the current program. Then program OVERLAY1 is merged into the current program from the default drive and control is transferred to line number 500.

Notes

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

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

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

**CHR\$
Function**

Purpose To convert an ASCII code to the character it represents. (ASCII codes are listed in Appendix B.)

Format CHR\$(x)

 x must be a numeric expression in the range 0 to 255.

Comments CHR\$ evaluates x, rounds it to an integer, and then returns the appropriate character. If x is not in the range 0 to 255, an "Illegal function call" error message is displayed.

CHR\$ is commonly used to send special characters to the display screen or the printer. For example, GWBASIC supports Greek, scientific, and graphic characters that are not represented by keys on the keyboard. To display or print these characters, the CHR\$ function is used with the ASCII code for these characters as its argument.

The CHR\$ function can also be used to send control functions like line feed, form feed, or carriage return from within your program.

Example 150 FOR INDEX = 1 TO 150
 160 LPRINT NAME\$(INDEX) TAB(20) ADDRESS\$(INDEX);
 170 LPRINT TAB(50) TOWN\$(INDEX) TAB(74) ZIP(INDEX)
 180 IF INDEX/50 = INT(INDEX/50) THEN LPRINT CHR\$(12)
 190 NEXT INDEX

In this example, the program is printing out a listing of names and addresses. The IF THEN statement in line 180 checks to see if a multiple of 50 names have been printed. If an even multiple of 50 has been printed, a form feed (ASCII code 12) is sent to the printer to advance the printer to the top of the next page.

Notes ASC is the complement of the CHR\$ function and converts a string character to its ASCII code.

**CINT
Function**

Purpose To convert a number to an integer by rounding the fractional portion.

Format CINT(x)

x must be a numeric expression in the range -32768 to 32767. If x is out of this range, an "Overflow" error message is displayed.

Comments The conversion is made by rounding x to the closest integer.

Examples PRINT CINT(45.67)
46
Ok
PRINT CINT(-1.75)
-2
Ok

Notes The FIX and INT functions can also be used to return integer values.

Also see the CSNG and CDBL functions for details on converting numbers to single- or double-precision.

**CIRCLE
Statement**

Purpose To draw an ellipse on the screen with the specified center and radius (graphics mode only).

Format CIRCLE (x,y),r[,<color>[,<start>,<end>[,<aspect>]]]

(x,y) are the x and y coordinates for the center of the ellipse. The coordinates may be entered in either absolute or relative coordinate form. (Refer to the Graphics Mode section of Chapter 4 for information on absolute and relative coordinates.)

r is the radius of the major axis of the ellipse in pixels (points).

color is a numeric expression specifying the color of the ellipse. In the medium resolution mode, <color> may be in the range 0 to 3. 0 selects the background color, while colors 1 to 3 select a color from the current palette (see the COLOR statement for information on selecting the palette). The default color is the foreground color, color number 3. In the high resolution mode, color 0 is black and color 1 is white (default).

start,end are numeric expressions in the range -2π to 2π (where $\pi = 3.141593$) indicating the start and end angles in radians. These angles allow you to specify where an ellipse will begin and end. Placing a negative sign in front of an angle will connect that end of the ellipse to the center point with a line, and the angle will be treated as if it were positive. The start angle may be less than the end angle. If <start> and <end> values are omitted, a complete ellipse is drawn (0 to 2π).

aspect is a numeric expression for the aspect ratio, or the ratio of the x radius to the y radius. The default aspect ratios are 5/6 for the medium-resolution mode, and 5/12 for the high-resolution mode. These default ratios were chosen to draw a circle on a standard screen with an aspect ratio of 4/3.

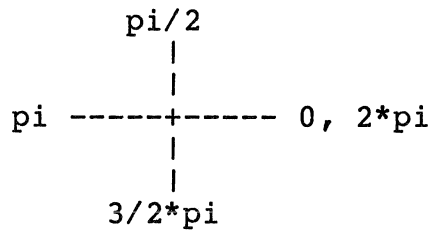
If the aspect ratio is less than one, r indicates the radius for the x axis. If it is greater than one, r is the radius for the y axis.

**CIRCLE
Statement**

Comments The CIRCLE statement can only be used in the graphics mode.

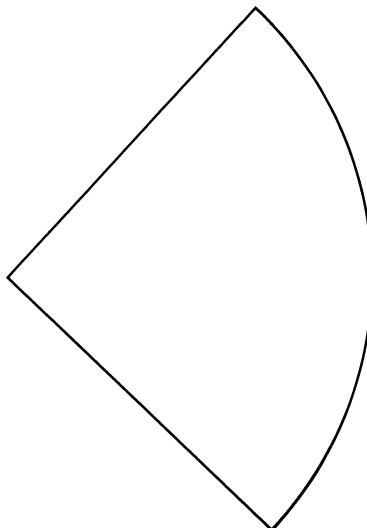
The CIRCLE statement is used to draw an ellipse, or arc segments of an ellipse, on the screen. A circle is a special case of an ellipse, and is drawn when the default <aspect> value is used.

The angles used for the <start> and <end> values are positioned around the ellipse in the standard mathematical way, with 0 in the three o'clock position and increasing in a counterclockwise direction as shown below.



Examples The following program segment shows how to draw a pie shaped wedge.

```
10 PI = 3.141593
20 SCREEN 1
30 CIRCLE (160,100),80,, -7*PI/4,-PI/4
```



**CIRCLE
Statement**

The following short program uses the CIRCLE statement several times to draw the curved lines for a cup.

```

10 PI = 3.141593
20 SCREEN 1 'MEDIUM RESOLUTION GRAPHICS
30 COLOR 0,1 'SET BACKGROUND = BLACK, PALETTE = 1
40 CLS 'CLEAR THE SCREEN
50 CIRCLE (140,30),50,,,,5/20 'DRAW TOP RIM OF CUP
60 CIRCLE (140,120),50,,PI,2*PI,5/20 'DRAW BOTTOM
70 LINE (90,30)-(90,120) 'DRAW LEFT SIDE
80 LINE (190,30)-(190,120) 'DRAW RIGHT SIDE
90 'DRAW HANDLE
100 CIRCLE (190,72),30,,3/2*PI,PI/2,1
110 CIRCLE (190,72),40,,3/2*PI,PI/2,1
120 PAINT (140,75),2,3 'COLOR CUP MAGENTA
130 PAINT (225,72),2,3 'COLOR HANDLE MAGENTA
140 PAINT (140,30),1,3 'COLOR INSIDE CYAN
150 END
    
```

Notes The last point referenced after an ellipse (or ellipse segment) is drawn, is the center point indicated by the x and y coordinates in the CIRCLE statement. That is, if you use relative coordinates in your next statement, you are giving directions relative to the center of the ellipse.

Points that are off the screen are not drawn by the CIRCLE statement.

**CLEAR
Statement**

Purpose To set all numeric variables to zero, all string variables to null, to close all open files, and optionally to set the end of memory and the amount of stack space.

Format CLEAR [,n][,m]

n is the maximum number of bytes to be set aside for the GWBasic workspace. This includes the interpreter workarea, program area, and data area. You would normally enter a value for n if you need to reserve space above GWBasic for a machine language subroutine. The default and maximum workspace size is 65,534 bytes (64K).

m sets aside stack space for GWBasic. The default stack size is 512 bytes or one-eighth of the available memory, whichever is smaller. If you use many nested GOSUB statements or FOR...NEXT loops, or use the PAINT statement to do complex fills, you might need to enter a value for m to reserve additional stack space.

Comments The CLEAR command is used to clear the memory area used for data storage without erasing the program currently in memory. The CLEAR command performs the following actions:

- * Closes all files
- * Clears all COMMON variables
- * Resets numeric variables to zero
- * Resets string variables to null
- * Resets arrays dimensioned with a DIM statement
- * Resets the stack and string space
- * Releases all disk buffers
- * Resets all DEF FN and DEF/SNG/DBL/STR statements
- * Turns off any sound that is running
- * Resets to Music Foreground
- * Resets PEN and STRIG to OFF

**CLEAR
Command**

Examples CLEAR

The data is cleared from memory without erasing the program currently in memory.

CLEAR,32768

The data is cleared from memory and the maximum workspace is set to 32768 bytes.

CLEAR ,,2000

The data is cleared from memory and the stack size is set to 2000 bytes.

CLEAR ,32768,2000

The data is cleared from memory, the maximum workspace is set to 32768 bytes, and the stack size is set to 2000 bytes.

Notes The ERASE statement can also be used to free memory by erasing arrays that will no longer be used.

**CLOSE
Statement**

Purpose To close I/O to a file or a device. The CLOSE statement is the complement to the OPEN statement.

Format CLOSE [[#]<file number>[, [#]<file number>]...]

file number is the number under which the file was opened in an OPEN statement.

Comments A CLOSE with no arguments closes all open files or devices.

The association between a file or device and its file number ends when the CLOSE statement is executed. The file or device may be reopened using the same or a different file number, or the file number may be used to open a different file or device.

A CLOSE writes the final buffer of output to a file or device opened for sequential output and places a ^Z end-of-file character (hex 26) at the end of the file.

A CLOSE statement is automatically executed when an END, NEW, RESET, SYSTEM, or RUN (without the R option) is executed. The STOP statement does not execute a CLOSE.

Examples 450 CLOSE #1,2

Closes the files or devices associated with file numbers 1 and 2.

790 CLOSE

Closes all open files or devices.

**CLS
Statement**

Purpose To erase or clear the display screen.

Format CLS

Comments In the text mode, CLS clears the active screen page to the background color and returns the cursor to the home position. The active screen page is set with the SCREEN statement, and the background is set with the COLOR statement.

In the graphics mode, CLS clears the screen buffer, and therefore the screen, to the background color. The last referenced point becomes the center of the screen. In the medium-resolution mode, this is 160,100; in the high-resolution mode, this is 320,100.

Example 10 CLS 'Clears the screen

Notes The SCREEN and WIDTH statements perform a CLS when they change the screen mode.

Pressing the <Ctrl>/<Home> key sequence also clears the screen.

COLOR**Statement (Text)**

Purpose To select the foreground, background, and border colors for the display.

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

foreground is a numeric expression in the range 0 to 31 representing the text color.

background is a numeric expression in the range 0 to 7 representing the background color.

border is a numeric expression in the range 0 to 15 representing the color for the border of the screen.

Comments If you are using a color monitor, the following sixteen colors are available for the foreground and border:

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-intensity White

You can make characters blink by adding 16 to the foreground color. Therefore, to display blinking Red characters, you would enter a foreground color of 20.

The background color may be chosen from any of the first 8 colors (colors 0 through 7).

If you are using a monochrome display (black and white), the COLOR statement allows you to use 16 shades of gray from black to high-intensity white. Colors 0 through 15 may be entered as foreground or border colors, and colors 0 through 7 may be entered as a background color. Adding 16 to the foreground color displays blinking characters.

COLOR**Statement (Text)**

Example The following simple program demonstrates the use of the COLOR statement to set different screen attributes.

```
10 SCREEN 0      'SET SCREEN TO TEXT MODE
20 CLS           'CLEAR SCREEN
30 COLOR 7,0,0   'SET SCREEN TO DEFAULT VALUES
40 PRINT "THIS IS A NORMAL LINE OF TEXT" : PRINT
50 COLOR 15      'SET HIGH INTENSITY
60 PRINT "THIS LINE IS IN HIGH INTENSITY" : PRINT
70 COLOR 23      'SET SCREEN TO BLINKING CHARACTERS
80 PRINT "THIS IS A BLINKING LINE OF TEXT" : PRINT
90 COLOR 0,7     'SET REVERSE VIDEO
100 PRINT "THIS LINE IS IN REVERSE VIDEO" : PRINT
110 COLOR 16,7   'SET BLINKING REVERSE VIDEO
120 PRINT "THIS LINE IS BLINKING REVERSE VIDEO" : PRINT
130 COLOR 7,0    'SET DISPLAY BACK TO DEFAULT VALUES
140 END
```

Notes An "Illegal function call" error message is displayed if a parameter is entered outside of the specified range. In this case, previous values are retained.

Setting the foreground color the same as the background color makes the characters invisible.

Any parameter can be omitted. If a parameter is omitted, the current value is retained. If the COLOR statement is entered without any parameters, an "Illegal function call" error message is displayed.

COLOR
Statement (Graphics)

Purpose To set the background color and current palette for the medium resolution graphics mode.

Format COLOR [<background>][, [<palette>]]

background is a numeric expression in the range 0 to 15 representing the background color. (See the text COLOR statement for a listing of the available colors.)

palette is a numeric expression selecting one of the two available palettes.

Comments The colors in each palette are assigned as follows:

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

Palette 0 is selected if <palette> evaluates to an even number. Palette 1 is selected if <palette> evaluates to an odd number.

If either of the parameters is omitted from the COLOR statement, that parameter retains the current value.

The COLOR statement is used to set the colors available to the PSET, PRESET, LINE, CIRCLE, PAINT, and DRAW statements.

Examples 10 SCREEN 1,0
 20 COLOR 1,3

Sets the background color to blue and selects palette 1. Text is printed in color 3 (white). To later change to palette 0, the following statement could be used:

120 COLOR ,4

Notes If a <background> value of 16 to 32 is entered, the intensity of the entire screen is increased.

An "Illegal function call" error message is displayed if a parameter is entered outside the range 0 to 255. If a value larger than 31 is entered for the background color, COLOR uses the value returned by formula:

<background> MOD 32

**COM(n)
Statement**

Purpose To enable or disable event trapping of communications activity on the specified communication channel.

Format COM(n) ON
COM(n) OFF
COM(n) STOP

n is the number of the communications channel (1 or 2).

Comments The COM(n) ON statement enables communications event trapping by an ON COM(n) statement. After executing the COM(n) ON statement, GWBasic checks between every statement to see if characters have come in to the specified communications channel. If characters have come in and a non-zero line was specified in the ON COM(n) statement, control is transferred to the specified line.

COM(n) OFF disables the communications event trapping. If an event takes place, it is not remembered.

COM(n) STOP disables the communications event trapping, but if an event occurs, it is remembered. The GOSUB statement is executed as soon as trapping is enabled with the next COM(n) ON statement.

Example 10 ON COM(1) GOSUB 500
20 COM(1) ON

Enables trapping of communications activity on channel 1.

Notes For additional information on communications event trapping, see the ON COM(N) Statement.

**COMMON
Statement**

Purpose To pass variables to a chained program.

Format COMMON <variable>[,<variable>]...

variable is the name of a variable to be passed to the chained program. Array variables are indicated by adding "()" after the variable name.

Comments The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement.

Example 100 COMMON COUNT,NAME\$(),AGE()
110 CHAIN "PROG3",10

This example chains program PROG3 from the default drive and passes the numeric variable COUNT, the string array NAME\$, and the numeric array AGE.

Notes If all the variables are to be passed from the current program to the chained program, use the ALL option with the CHAIN statement.

The "()" symbols after the array name passes arrays of all sizes and dimensions.

CONT
Command

Purpose To continue program execution after a <Ctrl>/<Break> has been entered, or a STOP or END statement has been executed.

Format CONT

Comments Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used in conjunction with the STOP statement for debugging. When execution is stopped, variables can be examined and changed using direct mode statements. Execution is resumed at the break point with the CONT command. CONT may be used to continue execution after an error has occurred.

Example

```

10 A = 10
20 PRINT A
30 STOP
40 PRINT A
RUN
  10
BREAK IN 30
Ok
A = 20
Ok
CONT
  20
Ok
    
```

Notes CONT is invalid if a program line is edited during the break.

The GOTO statement may be used from the direct mode to resume execution at a specified line number.

COS
Function

Purpose To return the trigonometric cosine value of the specified argument.

Format COS(x)

x is an angle in radians. To convert degrees to radians, multiply degrees by pi/180 (where pi = 3.141593).

Comments The calculation of COS is performed in single-precision. If the /D option was included when GWBASIC was started, COS is performed in double-precision.

Example 10 PI = 3.141593
 20 DEGREES = 360
 30 RADIANS = DEGREES * PI / 180
 40 PRINT COS(RADIANS)
 RUN
 1
 Ok

**CSNG
Function**

Purpose To convert a number to single-precision.

Format CSNG(x)

 x is any numeric expression.

Comments The CSNG function converts the result of numeric expression x to a single-precision number using the rules for numeric type conversion as described in Chapter 3.

Example 10 PI# = 3.14159265
 20 PRINT PI# CSNG(PI#)
 RUN
 3.14159265 3.141593

Notes See the CINT and CDBL functions for converting numbers to integer and double-precision.

**CSRLIN
Function**

Purpose To return the line (vertical) position of the cursor.

Format CSRLIN

Comments The CSRLIN function returns a value from 1 to 25 indicating the current line on the active page. (See the SCREEN statement for information on the active page.)

The CSRLIN function, along with the POS function, is used to store the current cursor position to allow the cursor to be moved to another part of the screen and then returned to the original position.

Example 50 ROW = CSRLIN 'STORE CURRENT LINE
60 COL = POS(0) 'STORE CURRENT COLUMN
70 INPUT "ENTER 5 DIGIT PART NUMBER";PARTNO\$
80 IF LEN(PARTNO\$) = 5 THEN 120
90 LOCATE ROW,COL : PRINT SPC(35)
100 LOCATE ROW,COL
110 GOTO 70

In this program segment, the POS and CSRLIN functions are used to store the location of the first character position of the INPUT prompt. If a 5-character response is not entered, line number 90 erase the prompt and the response. Line number 100, 110, and 70 repeat the prompt in the same place on the screen as the prompt originally appeared.

**CVI, CVS, CVD
Functions**

Purpose To convert string variables to numeric variables.

Format CVI(<2-byte string>)
 CVS(<4-byte string>)
 CVD(<8-byte string>)

Comments When numeric variables are stored in random access files, they are first converted to string variables. The CVI, CVS, and CVD functions are used to convert these string variables back to numeric variables. The CVI function is used to convert 2-byte strings back to integers. CVS converts 4-byte strings back to single-precision numbers. CVD converts 8-byte strings back to double-precision numbers.

Example 60 OPEN "DATAFILE" AS 1 LEN = 8
 70 FIELD #1,4 AS A1\$,4 AS A2\$
 80 GET #1,10
 90 FIRST = CVS(A1\$)
 100 SECOND = CVS(A2\$)

This example opens file DATAFILE as random file number one and reads in record number 10. The two 4-byte string variables in record 10 are converted to single-precision variables labeled FIRST and LAST.

Notes Refer to the MKI\$, MKS\$, and MKD\$ functions for information on converting numeric variable types to string variable types.

**DATA
Statement**

Purpose To store numeric and string constants that are accessed by the program's READ statements.

Format DATA <constant>[,<constant>]...

constant may be a string constant or a numeric constant in fixed-point, floating-point, integer, hex, or octal format. Numeric expressions are not allowed. String constants do not need to be surrounded by double quotation marks, unless they contain commas, colons, or significant leading or trailing spaces.

Comments DATA statements are non-executable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line, and any number of DATA statements may be included in the program. The data contained in DATA statements may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program. The READ statements access the DATA statements in the order in which they appear in the program.

The variable type (numeric or string) given in the READ statement must agree with corresponding constant in the DATA statement or a "Syntax error" error message is displayed.

Example 10 DATA 77,92,56,84,73
 20 DATA 89,95,51,75,80
 30 SUM = 0
 40 FOR INDEX = 1 TO 10
 50 READ SCORE
 60 SUM = SUM + SCORE
 70 NEXT INDEX
 80 AVERAGE = SUM / 10
 90 PRINT "THE AVERAGE IS" AVERAGE
 100 END
 RUN
 THE AVERAGE IS 77.2
 Ok

In this example, the FOR...NEXT loop is used to read the values from the DATA statements and add them up. The total is then divided by 10 to find the average.

Note The RESTORE statement can be used to restore the data in the DATA statements, allowing the data to be read again.

**DATE\$
Function**

Purpose To retrieve the current date.

Format DATE\$

Comments The DATE\$ function returns a ten-character string in the form mm-dd-yyyy where:

 mm represents the current month (01 through 12)

 dd represents the current day (01 through 31)

 yyyy represents the current year (1980 through 2099)

 The DATE\$ function returns the date set by the most recent DATE\$ statement or TeledOS DATE command.

Example PRINT DATE\$
 01-01-1980
 Ok

**DATE\$
Statement**

Purpose To set the current date known to the operating system.

Format DATE\$ = x\$

x\$ is a string expression representing the current date in one of the following formats:

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

where:

mm is a one- or two-digit number from 1 to 12 representing the current month.

dd is a one- or two-digit number from 1 to 31 representing the current day of the month.

yy is a two-digit number from 80 to 99 representing the current year (the 19 is assumed).

yyyy is a four-digit number from 1980 to 2099 representing the current year.

Comments The DATE\$ statement performs a function similar to the TeleDOS DATE command.

Example DATE\$ = "7-1-83"
 Ok
 PRINT DATE\$
 07-01-1983
 Ok

**DEF FN
Statement**

Purpose To name and define a function that you create.

Format DEF FN<name>[(**<variable>**[,**variable**]...)] = **<function>**

name is a valid variable name. This name, preceded by FN, becomes the name of the function.

variable is a variable name representing a dummy variable that is used to define the function. When the function is later used, the variable holding that position in the variable list is substituted everywhere the dummy variable exists in the function (see the following example).

function is an expression returning a value of the type indicated by the function **<name>**. Variables used to define the function may or may not appear in the variable list. If they do, the actual value of the variable is supplied when the function is called. Otherwise, the current value of the variable is used.

Comment The DEF FN statement allows you to define a commonly used calculation or expression as a function. You can then use the function call in your program instead of having to reenter the expression each time. The DEF FN statement may be used to define either numeric or string functions. The function type is determined by the type of the **<name>** argument.

The DEF FN statement must define the function in a program before the function can be called. If a function is called before it is defined, an "Undefined user function" error message is displayed.

Examples 10 PI = 3.141593
 20 'DEFINE FNCIRCUM TO FIND THE CIRCUMFERENCE
 30 'OF A CIRCLE OF A GIVEN RADIUS
 40 DEF FNCIRCUM(R) = 2 * PI * R
 50 INPUT "Enter the radius: ",RADIUS
 60 PRINT "The circumference is equal to";
 70 PRINT FNCIRCUM(RADIUS)
 80 END
 RUN
 Enter the radius: 1
 The circumference is equal to 6.283186
 Ok

**DEF FN
Statement**

In this example, line 40 defines the function FNCIRCUM for calculating the circumference of a circle from its radius. The function multiplies two times the variable PI, times the value passed as R when the function is called. Line number 70 calls the function and tells it to use the variable RADIUS as the value for R.

```

10 'DEFINE FNHYPOT TO FIND THE HYPOTENUSE OF A
20 'RIGHT TRIANGLE
30 DEF FNHYPOT(A,B) = SQR(A^2 + B^2)
40 PRINT "Enter the two sides of a right triangle ";
50 INPUT S1,S2
60 PRINT "The hypotenuse is equal to" FNHYPOT(S1,S2)
70 END
RUN
Enter the two sides of a right triangle ? 3,4
The hypotenuse is equal to 5
    
```

In this example, line number 30 defines the function FNHYPOT for calculating the hypotenuse of a right triangle given the two sides. Dummy variables A and B are used in the function definition to represent the two sides. Line number 60 calls the function FNHYPOT and tells it to use variables S1 and S2 as the values for A and B respectively.

Notes If the value type returned by the function does not match the variable name type used to identify the function, a "Type mismatch" error message is displayed.

**DEF SEG
Statement**

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

Format DEF SEG [=<address>]

address is a numeric expression returning an unsigned integer in the range -32768 to 65535 representing a 16-byte paragraph boundary. If <address> is negative, DEF SEG uses the value (65536 + <address>). The value entered is shifted left four bits (multiplied by 16) to indicate the beginning address of the segment.

Comments The DEF SEG statement is used to set the segment address to be used in subsequent BLOAD, BSAVE, CALL, CALLS, or POKE statements, or by a USR or PEEK function. These statements and functions then indicate an address by giving an offset into this segment.

GWBasic initially sets this segment address to GWBasic's Data Segment (DS). The GWBasic Data Segment indicates the beginning of your user workspace in memory. If the DEF SEG statement is entered without an <address> parameter, the segment defaults to the GWBasic Data Segment.

Example 10 SCREEN 0 : CLS 'CLEAR THE SCREEN
 20 'SET THE SEGMENT TO VIDEO MEMORY AT HEX B8000
 30 DEF SEG = &HB800
 40 'PRINT SMILEY FACES ON LINE 1
 50 FOR INDEX = 0 TO 160 STEP 4
 60 POKE INDEX,1
 70 NEXT INDEX
 80 'RESET SEGMENT TO THE GWBASIC DATA SEGMENT
 90 DEF SEG
 100 PRINT
 110 END

In line number 20, the DEF SEG statement is used to set the segment to the video memory at hex B8000. The FOR NEXT loop in lines 50 through 70 POKE ASCII character 1 (a smiley face) into every other character position of the top line of the display memory. Line number 90 resets the segment to GWBasic's Data Segment.

**DEF SEG
Statement**

Notes If an <address> value is entered outside of the -32768 to 65535 range, an "Overflow" error message is displayed and the current value is retained.

DEF and SEG must be separated by a space or the statement will be interpreted as assign value <address> to numeric variable DEFSEG.

**DEFtype
Statements**

Purpose To declare variable types as integer, single-precision, double-precision, or string.

Format DEF<type> <letter>[-<letter>][,<letter>[-<letter>]]...
 type is INT for integer, SNG for single-precision, DBL for double-precision, or STR for string.
 letter is a letter of the alphabet (A-Z).

Comments The DEFtype statement can be used to declare variables starting with the designated letter or letters as belonging to that variable type. Variable names with these letters will not need the type declaration character (% , ! , # , \$) to specify its type. An added variable type declaration character takes precedence over the DEF<type> statement.

DEF<type> statements should be located at the beginning of your program before any of the declared variables are used.

Example 10 DEFSTR A-D
 20 DEFINT I-N
 30 ALPHA = "CAT"
 40 I = 3.4
 50 LAST! = 12.7
 60 PRINT ALPHA I LAST!
 RUN
 CAT 3 12.7
 Ok

In this example, line 10 defines variables starting with the letters A-D as string variables. Therefore the variable ALPHA does not need the \$ type declaration character. Line 20 defines all variables starting with the letters I-N as integer type. When 3.4 is assigned to variable I in line 40, the value is rounded to an integer before it is stored. The variable LAST! in line 50 contains the ! single-precision type declaration character, and therefore overrides the integer definition of line 20.

**DEF USR
Statement**

Purpose To specify the starting address of an assembly language routine that will later be called by the USR function.

Format DEF USR[n]=<offset>

 n is a number from 0 to 9 identifying the USR routine whose starting address is being specified. If omitted, DEF USR0 is assumed.

 offset is an integer expression in the range -32768 to 65535 indicating the offset into the current segment (see the DEF SEG statement). If <offset> is negative, a value of (65536 + <offset>) is used. The starting address is obtained by adding the offset value to the segment value.

Comments Any number of DEF USR statements may appear in a program to redefine a subroutine starting address.

Example 20 DEF SEG = 0
 30 DEF USR1 = 24000
 .
 .
 150 ANSWER = USR1(Y)

This example sets the starting address for subroutine one (1) at absolute memory address 24000.

**DELETE
Command**

Purpose To delete program lines from the program currently in memory.

Format DELETE [<start line>][-<end line>] or

DELETE [<start line>-]

start line is the line number of the first line to be deleted.

end line is the line number of the last line to be deleted.

Comments The DELETE command deletes the specified line or range of lines from the program currently in memory and then returns to the command level.

If only one line number is listed, only that line is deleted. If both a start and end line are entered, the start line, the end line, and any line number in between is deleted.

Entering a dash and a line number will delete all program lines from the beginning of the program up to and including the entered line number. Entering a line number and a trailing dash deletes that line number and all subsequent line numbers in the program.

Examples DELETE 40

Deletes line number 40 from the program in memory.

DELETE 40-100

Deletes all program lines in the range 40 through 100, inclusive.

DELETE -40

Deletes all program lines in the range 0 to 40, inclusive.

DELETE 40-

Deletes all program lines from line 40 to the end of the program.

Notes If you try to delete a line by only entering a <start line>, and the line number does not exist in memory, an "Illegal function call" error message is displayed.

DIM
Statement

Purpose To specify the maximum values for array variable subscripts and to allocate enough memory storage for the arrays.

Format DIM <variable>(<subscripts>)
[,<variable>(<subscripts>)]...

variable is a valid array variable name.

subscripts is a list of numeric expressions, separated by commas, indicating the subscript size for each dimension of the array.

Comments Array variables are set with a default maximum subscript value of ten (10) for each dimension in the array. In Option Base 0, this allows eleven item per dimension (using subscripts 0 through 10). To use larger subscripts, and therefore more elements per dimension, the DIM statement must be used.

The DIM statement sets the largest subscript value that can be used in each dimension of the array. The DIM statement initially sets each element of a numeric array equal to zero, and each element of a string array to the null string.

The maximum number of dimensions per array is 255, and the largest subscript size per dimension is 32767. These numbers are in reality limited by the amount of memory available and the length of a program line.

A "Subscript out of range" error message is displayed if a subscript larger than the dimensioned value is used. This also applies to using a subscript larger than ten (10) in an array that has not been dimensioned using a DIM statement.

A "Duplicate Definition" error message is displayed if you try to dimension an array more than once. This message is also displayed if you try to dimension an array after you have already assigned a value to one of its elements.

**DIM
Statement**

Example 10 DIM SCORES(20)
 20 SUM = 0
 30 FOR INDEX = 1 TO 20
 40 PRINT "Enter score number" INDEX " ";
 50 INPUT SCORES(INDEX)
 60 SUM = SUM + SCORES(INDEX)
 70 NEXT INDEX
 80 PRINT
 90 PRINT "Your average score is" SUM / 20

This example dimensions the one-dimensional numeric array SCORES to a maximum subscript value of 20. The array is then used to store twenty scores.

Notes The ERASE statement can be used to erase an array, allowing you to dimension it again.

**DRAW
Statement**

Purpose To draw an object on the screen in the graphics mode.

Format DRAW <string>

 string is a string expression using subcommands to describe the object to be drawn.

Comments The DRAW statement is used to draw objects using a graphics definition language. The graphics definition language subcommands are used in the <string> expression to define the object to be drawn. These subcommands describe motion (up, down, left, right), color, angle, and scale factor.

Each of the following subcommands begins movement from the current graphics position. The current graphics position is the last point referenced by a graphics command. This is usually the coordinate of the last graphics point plotted with a command. The current position defaults to the center of the screen when a program is started.

U[n] Move up
D[n] Move down
L[n] Move left
R[n] Move right
E[n] Move diagonally up and right
F[n] Move diagonally up and left
G[n] Move diagonally down and left
H[n] Move diagonally down and right

 n indicates the distance to move. The number of points moved is n times the scaling factor (set by the S subcommand). If n is omitted, a default value of 1 is used.

Mx,y Move absolute or relative. If x is preceded by a plus (+) or minus (-), x and y are added to the coordinates of the current graphics position and a line is drawn from the current position to the new position. Otherwise, a line is drawn to point x,y from the current position.

**DRAW
Statement**

The following two prefix commands may precede any of the above movement commands, and affect only the command it precedes:

- B Move but don't plot any points.
- N Move but return to the original position when done (the last point referenced or current graphics position is not changed by the command).

The following additional commands are also available:

- An Set angle n. n may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270. Figures rotated 90 or 270 degrees are scaled so they will appear the same size as with 0 or 180 degrees on a monitor with the standard aspect ratio of 4/3.
- TAn Turn angle n. n is in degrees and can range from -360 to +360. If n is positive, the angle rotates counterclockwise. If n is negative, the angle rotates clockwise. An "Illegal function call" error message is displayed if a value outside of the range -360 to +360 is entered.
- Cn Set color n. n may range from 0 to 3 in medium resolution, and 0 to 1 in high resolution. In medium resolution, n selects the background color (0) or one of the three colors (1-3) from the current palette. The default value is the foreground color, color 3. In high resolution, 0 selects black, and 1 selects white (the default value).
- Sn Set scale factor. n may range from 1 to 255. n divided by 4 gives the actual scale factor. The scale factor multiplied by the distances given with U, D, L, R, or relative M commands gives the actual distance traveled. The default value for n is 4, giving a default scale factor of 1.

**DRAW
Statement**

X<string variable>;

Execute substring. This powerful command allows you to execute a second substring from within a string, much like GOSUB in BASIC. This allows you to create a string of commands more than 255 characters long, or to define portions of an object separate from the entire object. A semicolon (;) is required after the string variable name when using the X subcommand.

P<color>,<border>

Paint using color <color> within a border of color <border>. The P subcommand is a fill command, painting the area enclosed by the specified border color with the specified color. <color> and <border> can range from 0 to 3 in medium resolution, 0 indicating the background color, and 1 to 3 indicating the three colors of the current palette. In high resolution, <color> and <border> can be 0 or 1, 0 indicating black, and 1 indicating white.

The numeric arguments n, x, and y used in the above subcommands can be expressed as numeric constants, like **123**, or as numeric variables in the form **=<variable>;**, where <variable> is the name of a numeric variable and the = sign and the semicolon (;) are required.

Examples The following three examples show three ways to draw the same box on the screen.

```
10 SCREEN 1
20 DRAW "U40R48D40L48"
30 END
```

```
10 SCREEN 1
20 HORIZ = 48
30 VERT = 40
40 DRAW "U=VERT;R=HORIZ;D=VERT;L=HORIZ;"
50 END
```

**DRAW
Statement**

```

10 SCREEN 1
20 LEFTSIDE$ = "U40"
30 TOP$ = "R48"
40 RIGHTSIDE$ = "D40"
50 BOTTOM$ = "L48"
60 BOX$ = LEFTSIDE$ + TOP$ + RIGHTSIDE$ + BOTTOM$
70 DRAW "XBOX$;"
80 END
    
```

The following example shows the use of the TA subcommand to create a starburst pattern.

```

10 SCREEN 1
20 CLS
30 FOR ANGLE = 0 TO 360 STEP 10
40 DRAW "TA=ANGLE;NU60"
50 NEXT ANGLE
60 END
    
```

Notes

The aspect ratio of the monitor determines the spacing of points on horizontal, vertical, and diagonal lines. The aspect ratio of a standard monitor is 4/3, indicating that the horizontal axis of the screen is 4/3 as long as the vertical axis. This information is needed to create horizontal and vertical lines of equal length.

In medium resolution, with 320 horizontal pixels by 200 vertical pixels, a 1/1 aspect ratio would require 8 horizontal pixels to be the same length as 5 vertical pixels. With a 4/3 aspect ratio, you must multiply the vertical length by 3/4 to get an equal horizontal length; therefore, to find an equal horizontal length (x) for a known vertical length (y), use the formula:

$$x = 6/5 * y \quad (8/5 * 3/4 = 6/5)$$

To find an equal vertical length (y) for a known horizontal length (x), use:

$$y = 5/6 * x$$

In high resolution, with 640 horizontal pixels by 200 vertical pixels, the formulas to find equivalent lengths are:

$$x = 12/5 * y \quad \text{and} \quad y = 5/12 * x$$

EDIT

Command

-
- Purpose** To enter the edit mode at a specified line.
- Format** EDIT <line number>
- line number is a valid line number in the program currently in memory.
- Comments** The EDIT command displays the specified line and then positions the cursor under the first digit of the line number. You can then use the editing keys of the program editor to make any needed changes.
- A period (.) can be entered in place of a line number to refer to the current line (the last line edited or referenced). If a period (.) is used, at least one space must separate the T of EDIT and the period or a "Syntax error" error message is displayed.
- Example** EDIT 40
 40 RIGHTSIDE\$ = "D40"
- Note** The LIST command may also be used to display program lines for editing.

**END
Statement**

Purpose To terminate program execution, close all files, and return control to the command level.

Format END

Comments END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a "Break in line nnnnn" message to be displayed. An END statement at the end of a program is optional.

Example 520 IF K>1000 THEN END

In this example, if the value of numeric variable K is greater than 1000 the program terminates; otherwise, execution continues with the next program line.

**EOF
Function**

Purpose To test for the end-of-file condition.

Format EOF(<file number>)

file number represents the number under which the file you are checking was opened for input in the OPEN statement.

Comments The EOF function can be used to avoid trying to read past the end of a sequential file. The EOF function returns -1 (true) if the end of the file has been reached; otherwise, a 0 (false) value is returned.

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

Example 10 OPEN "B:SCORES.DAT" FOR INPUT AS #1
 20 INDEX = 0
 30 DIM SCORE(50)
 40 WHILE (NOT EOF(1)) AND (INDEX < 50)
 50 INDEX = INDEX + 1
 60 INPUT #1,SCORE(INDEX)
 70 WEND
 80 CLOSE

This example program segment opens up the sequential file SCORES.DAT on drive B to read in scores. The array SCORE is dimensioned to hold up to 51 scores. The scores are read in using the WHILE...WEND loop in lines 40 through 70. The counter INDEX is used to place the scores in the array and to keep track of the total number of scores read. The WHILE...WEND loop continues to read in values until either the end of the file is reached (NOT EOF(1) becomes false) or the maximum dimension of the array is reached.

**ERASE
Statement**

Purpose To eliminate arrays from memory.

Format ERASE <array name>[,<array name>]...

array name is the name of a program array you want
to erase from memory.

Comments The ERASE statement may be used to free program memory
by erasing the space allocated to an array or arrays.
The ERASE statement can also be used if you want to
redimension an array. If you try to redimension an
array without first erasing it, a "Duplicate
definition" error message is displayed.

Example

```
      .  
      .  
      .  
30 DIM A(30),B(30)  
      .  
      .  
      .  
250 ERASE A,B  
260 DIM B(50)  
      .  
      .  
      .
```

In this example, numeric arrays A and B are originally dimensioned to 30. Line 250 erases arrays A and B from memory, and then line 260 redimensions array B to 50. Any values originally placed in array B are lost when line 250 is executed.

**ERR and ERL
Functions**

Purpose	To return an error code and the line number the error occurred in.
Format	ERR ERL
Comments	ERR and ERL are used by error handling routines to determine the type of error that has occurred. ERR contains the error code for the error and ERL contains the line number of the line in which the error was detected. ERR and ERL are usually used in IF...THEN statements to direct program flow in the error handling routine.

ERR and ERL can not appear on the left side of the equal sign in a LET (assignment) statement.

When ERL is used in a IF...THEN statement, use the form:

IF ERL=line number THEN ...

so that the line number can be renumbered by the RENUM command. If ERL appears on the right side and the line number on the left, the line number is not renumbered by the RENUM statement.

The error code numbers returned by ERR are listed in Appendix A.

Example The following program segments show how the ERR and ERL functions can be used to avoid having a program terminate in an error message. Two error handling routines are included to cover the cases when the user forgets to place the data disk in drive B, or the wrong data disk is placed in drive B. In these cases, a message is displayed telling the user how to correct the problem and then letting them continue with the program.

**ERR and ERL
Functions**

```
10 ON ERROR GOTO 1000
.
.
120 OPEN "B:INFO.DAT" FOR INPUT AS #1
.
.
1000 'ERROR HANDLING ROUTINES
1010 IF ERR = 53 GOTO 1050 'WRONG DISK
1020 IF ERR = 71 GOTO 1100 'NO DISK
1030 PRINT "ERROR NUMBER" ERR "IN LINE NUMBER" ERL
1040 END
1050 'FILE NOT FOUND
1060 PRINT "FILE INFO.DAT WAS NOT FOUND ON DRIVE B"
1070 PRINT "PLACE THE CORRECT DATA DISK IN DRIVE B"
1080 GOTO 1200
1100 'NO DISK IN DRIVE B
1110 PRINT "THERE IS NO DATA DISK IN DRIVE B"
1120 PRINT "OR THE DRIVE DOOR IS OPEN"
1130 PRINT
1140 PRINT "PLACE THE DATA DISK IN DRIVE B"
1150 PRINT "AND CLOSE THE DRIVE DOOR"
1200 INPUT "THEN PRESS RETURN",A$ : RESUME
```

Notes If an error occurs when a direct mode statement is executed, ERL will contain 65535.

**ERROR
Statement**

Purpose To simulate the occurrence of a BASIC error, or to allow error codes to be defined by the user.

Format ERROR n

n is a numeric expression in the range 0 to 255.

Comments If the value of n equals an error code already assigned by GWBASIC (see Appendix A), the ERROR statement simulates the occurrence of that error. If an error handling routine has been defined using the ON ERROR statement, program control is transferred to the error handling routine. Otherwise, the corresponding error message is displayed and program execution is terminated.

To define your own error code, use a value that is greater than any used by the GWBASIC error codes. (We suggest you use the highest available values so compatibility may be maintained when more error codes are added to GWBASIC.) Your new error code may then be handled in an error handling routine like the GWBASIC error codes.

If an ERROR statement specifies a code for which no error message has been defined, and you don't handle the error with an error handling routine, the "Unprintable error" error message is displayed.

Examples 10 S = 2
 20 ERROR S
 30 END
 RUN
 Syntax error in 20
 Ok

 ERROR 2
 Syntax error

The above examples show how to simulate a syntax error in both the indirect and direct modes.

**ERROR
Statement**

The next example shows segments of a game program which defines a bet of over 5000 as error code 210. The error handling routine displays a message and then returns control to the INPUT statement.

```

10 ON ERROR GOTO 400
   .
   .
110 INPUT "WHAT IS YOUR BET";B
120 IF B > 5000 THEN ERROR 210
   .
   .
400 'ERROR HANDLING ROUTINES
410 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
420 IF ERR = 210 AND ERL = 120 THEN RESUME 110
   .
   .

```

**EXP
Function**

Purpose To calculate the exponential function (e to the power of x).

Format EXP(x)

 x is a numeric expression less than or equal to 88.02968.

Comments The EXP function returns the value of e raised to the x power, where e is the base for natural logarithms (2.718282). If x is greater than 88.02968, an "Overflow" error message is displayed.

Example PRINT EXP(1)
 2.718282
 Ok

**FIELD
Statement**

Purpose To allocate space for variables in a random file buffer.

Format FIELD [#]<file number>,<width> AS <variable>
 [,<width> AS <variable>] ...

 file number represents the number under which the file was opened in the OPEN statement.

 width is a numeric expression indicating the number of bytes (character positions) to allocate for the field identified by the following variable name.

 variable is a valid string variable name used to identify a record field.

Comments The FIELD statement is used to format the random file buffer. It defines the variables used to transfer data to and from the buffer. The FIELD statement must be executed after the random access file is opened and before a GET or PUT statement for that file is executed.

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

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed for a file remain in effect. Each new FIELD statement redefines the buffer from the first byte; therefore, multiple field definitions may exist for the same data in the buffer.

Examples FIELD 1,20 AS N\$,10 AS ID\$,40 AS ADD\$

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

**FIELD
Statement**

```

10 OPEN "B:PHONELST" AS #1 LEN = 35
20 FIELD #1,2 AS COUNT$,33 AS DUMMY$
30 FIELD #1,25 AS NAME$,10 AS PHONENBR$
40 GET #1,1
50 TOTAL% = CVI(COUNT$)
60 PRINT "NAME" TAB(30) "PHONE NUMBER"
70 PRINT
80 FOR I = 2 TO TOTAL%
90 GET #1, I
100 PRINT NAME$ TAB(30) PHONENBR$
110 NEXT I
120 CLOSE
130 END
    
```

This example shows the use of multiple FIELD statements for the same file. The file PHONELST is used to store names and phone numbers, with the first record storing the number of the last record with valid information. Lines 20 and 30 define the two variable formats that will be used to transfer data from the buffer. Line 40 removes the last record number from record one (1) using the FIELD format of line 20. Line 50 uses the CVI function to convert the COUNT\$ string back to an integer containing the number of the last valid record. The FOR...NEXT loop in lines 80 through 110 is used to print out the names and phone numbers stored in the remaining records using the FIELD format of line 30.

Note Do not use a variable name defined in a FIELD statement on the left side of a LET statement or in a INPUT statement. Once a variable name is defined in a FIELD statement, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

FILES

Command

Purpose To print the names of the files residing in the specified directory of a disk. The FILES command is similar to the TeleDOS DIR command.

Format FILES [<filespec>]

filespec is a string expression indicating a file. If a filespec is omitted, all the files in the current directory on the default disk are displayed.

Comments All files with filenames matching a filename in filespec are displayed. The filename may contain the (?) and (*) wildcard characters in the name or extension.

If a drive designation and path are included as part of the filespec, all the files matching the filename in the specified directory on that drive are displayed. If the drive designation is omitted, the TeleDOS default drive is used. If a path is omitted, the current directory is used.

Examples FILES

Displays all files in the current directory on the TeleDOS default drive.

FILES "*.BAS"

Displays all files on the default drive with a filename extension of .BAS.

FILES "B:*.*)" or FILES "B:"

Displays all files on drive B.

FILES "B:\SALES"

Displays the subdirectory file SALES on drive B.

FILES "B:\SALES\"

Displays all files in the subdirectory SALES on drive B.

Notes Subdirectory files displayed in a directory listing are denoted by <DIR> after the directory name.

**FIX
Function**

Purpose To truncate a number to an integer.

Format FIX(x)

 x is any numeric expression.

Comments FIX strips all the digits to the right of the decimal point and returns an integer value equal to the digits to the left of the decimal point.

Examples 10 DATA -3.45,-0.79,0.50,58.75
 20 FOR INDEX = 1 TO 4
 30 READ NUMBER
 40 PRINT FIX(NUMBER)
 50 NEXT INDEX
 60 END
 RUN
 -3
 0
 0
 58
 Ok

Note Also see the INT and CINT functions for alternative methods of returning integers.

**FOR...NEXT
Statements**

Purpose To allow a series of instructions to be performed as a
 loop a given number of times.

Format FOR <variable> = x TO y [STEP z]
 .
 .
 .
 NEXT [<variable>][,<variable>] ...

variable is a valid integer or single-precision
 variable name to be used as a counter to keep
 track of the number of times the loop is
 repeated.

x is a numeric expression representing the
 initial value of the counter.

y is a numeric expression representing the
 final value of the counter.

z is a numeric expression representing the
 increment to use to change the counter from
 the initial value to the final value. z may
 be either positive or negative.

Comments Program execution of a FOR...NEXT loop proceeds as
 follows:

- 1) When the FOR statement is executed, the counters
 final value is established and then the counter
 variable is set equal to the initial value.
- 2) The initial value is compared to the final value.
 If the initial value does not exceed the final
 value, program control continues with the first
 statement after the FOR statement. If the initial
 value exceeds the final value, program control is
 transferred to the first statement after the NEXT
 statement.
- 3) Program control continues in order from the
 statement after the FOR statement until the NEXT
 statement is encountered.

**FOR...NEXT
Statement**

- 4) When the NEXT statement is encountered, the counter is incremented by the step value and again compared with the final value. If the new counter value does not exceed the final value, program control is transferred back to the first statement after the FOR statement. If the new counter value now exceeds the final value, control continues with the first statement after the NEXT statement.

If STEP is not specified, the increment defaults to one. If STEP is negative, the counter is decremented from the initial value to the final value and the loop is executed until the counter is less than the final value.

The counter must be an integer or single-precision numeric constant. If a double-precision numeric constant is used, a "Type mismatch" error message is displayed.

The statements between the FOR and NEXT statements are skipped if the initial value of the counter is greater than the final value and step is positive, or the initial value is less than the final value and step is negative.

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within 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, a single NEXT statement with the variable names may be used.

The level or number of nested FOR...NEXT loops is only limited by the amount of stack space.

The variable name(s) may be omitted from the NEXT statement, in which case the NEXT statement is matched to the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is displayed.

**FOR...NEXT
Statements**

Examples 10 FINISH = 5 : TOTAL = 10
 20 FOR INDEX = 1 TO FINISH
 30 PRINT INDEX;
 40 TOTAL = TOTAL + 10
 50 PRINT TOTAL
 60 NEXT

RUN
 1 20
 2 30
 3 40
 4 50
 5 60
 Ok

10 FINISH = 0
 20 FOR INDEX = 1 TO FINISH
 30 PRINT INDEX
 40 NEXT INDEX
 50 PRINT "COMPLETED"

RUN
 COMPLETED
 Ok

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

10 INDEX = 5
 20 FOR INDEX = 1 TO INDEX + 5
 30 PRINT INDEX;
 40 NEXT

RUN
 1 2 3 4 5 6 7 8 9 10
 Ok

In this example, the loop executes ten times. The final value for the loop variable is set before the initial value is set.

**FRE
Function**

Purpose To return the number of bytes in memory that are not being used by GWBASIC.

Format FRE(x) or FRE(x\$)

 x and x\$ are dummy variables.

Comments The FRE function used with a dummy numeric variable, such as FRE(0), returns the number of free bytes in the GWBASIC user workspace. This number does not include the memory area reserved for the interpreter workarea.

 GWBASIC manipulates strings dynamically, allowing string lengths to vary. Because of this, string space may become fragmented. Using the FRE function with a dummy string variable, such as FRE(""), forces a housecleaning before returning the number of free bytes. The housecleaning compresses the string data and frees unused areas of memory that had been used for strings.

 GWBASIC automatically performs the housecleaning procedure when all free memory has been used up. To reduce the delay for each housecleaning, you might want to use FRE("") periodically.

Example PRINT FRE(0)
 61706
 Ok

Notes If the above direct mode command is entered immediately after GWBASIC is started, the value returned should equal the number of free bytes listed in the GWBASIC version message.

GET

Statement (Files)

Purpose To read a record from a random access disk file into a random buffer.

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

file number is the number under which the file was opened in the OPEN statement.

record number is a numeric expression representing the record to be read from the file. <record number> must be in the range 1 to 32767. If <record number> is omitted, the next record (after the last GET) is read into the buffer.

Comments The GET statement is used to read a random access file record into the file buffer. Because GWBASIC and TeleDOS block as many records as possible into a 512 byte sector, the GET statement may not always have to physically read the record in from the diskette. Once the record is in the buffer, references to the variables defined in the FIELD statement are used to read the data from the buffer.

The GET statement may also be used for input from communications files. In this case <record number> represents the number of bytes to read from the communications buffer. This value cannot exceed the maximum value set in the OPEN COM statement.

Example 10 OPEN "B:NAMES.DAT" AS #1 LEN = 25
 20 FIELD #1,25 AS NAMES\$
 30 FOR INDEX% = 1 TO 10
 40 GET #1,INDEX%
 50 PRINT NAMES\$
 60 NEXT INDEX%
 70 CLOSE

In this example, line 10 opens file NAMES.DAT on drive B for random access. Line 20 defines the length of field variable NAMES\$ at 25 bytes. The FOR...NEXT loop in lines 30 through 60 read in the first ten records and print the name stored in each record.

**GET
Statement (Graphics)**

Purpose To read points from an area on the screen and store the information in an array.

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

(x1,y1)-(x2,y2) represent the coordinates of opposite corners of a rectangle on the display screen. (The rectangle is defined the same way as the rectangle drawn by the LINE statement using the ,B option.)

array name is a valid dimensioned numeric array name used to hold the information from the screen.

Comments The GET statement is used in the graphics mode to read a portion of the screen into an array so the image can be transferred to another part of the screen using the PUT statement. The graphics GET and PUT statements provide an easy method of creating high speed object motion for animation.

The GET statement stores the colors of the points within a specified rectangle of the screen. The rectangle is specified by entering the coordinates of two opposite corners. For example, the coordinates (0,0) and (19,39) would specify a rectangle 20 points wide by 40 points high enclosed by corners with coordinates (0,0), (19,0), (19,39), and (0,39).

The array is used as a means of storing and referencing the graphic image. The array must be numeric, but can be of any precision. The data is stored in the array with the x dimension in the first two bytes, the y dimension in the next two bytes, and the screen data in the following bytes. Eight bits are stored in each byte of memory, with each row of points being left justified on a byte boundary. If there are less than a multiple of eight bits stored, the rest of the byte is filled with zeros. The following formula can be used to calculate the number of bytes needed to store the image:

$$4 + \text{INT}((x * \text{bitsperpixel} + 7) / 8) * y$$

GET

Statement (Graphics)

- x is the number of points in the horizontal direction (x direction).
- y is the number of points in the vertical direction (y direction).
- bitsperpixel is the number of bits per pixel. In medium resolution this value is 2, in high resolution 1.

For example, if you wanted to GET a portion of the screen 10 points wide by 12 points high, you would need $4 + \text{INT}((10 \times 2 + 7) / 8) \times 12$, or 40 bytes. The number of bytes per element in an array are:

- * 2 bytes per integer
- * 4 bytes per single-precision
- * 8 bytes per double-precision

Therefore, you could dimension an integer array to 20 elements to hold the data.

Example The following example uses the GET and PUT statements to move the TeleVideo logo (drawn in lines 60 through 80) from the left edge of the screen to the center.

```

10 'SET UP SCREEN IN GRAPHICS MODE
20 SCREEN 1,0 : COLOR 0,1 : CLS
30 'DIMENSION A TO HOLD A 71 x 76 POINT AREA
40 DIM A(343) 'SINGLE-PRECISION ARRAY
50 'DRAW LOGO
60 DRAW "BM0,35D30M30,85U30M0,35" 'LEFT SIDE
70 DRAW "BM5,30M35,10M65,30M35,50M5,30" 'TOP
80 DRAW "BM70,35D30M40,85U30M70,35" 'RIGHT SIDE
90 'PUT LOGO IN ARRAY A
100 GET (0,10)-(70,85),A
110 'ERASE LOGO
120 PUT (0,10),A
130 'USE FOR...NEXT LOOP TO MOVE LOGO
140 FOR I = 5 TO 125 STEP 20
150 PUT (I,10),A 'DRAW LOGO
160 FOR J = 1 TO 100 : NEXT J 'DELAY LOOP
170 PUT (I,10),A 'ERASE LOGO
180 NEXT I
190 PUT (130,10),A 'DRAW LOGO IN CENTER OF SCREEN
200 'FILL IN SIDES
210 PAINT (165,15),2,3 'FILL TOP
220 PAINT (150,65),2,3 'FILL LEFT SIDE
230 PAINT (180,65),2,3 'FILL RIGHT SIDE
240 LOCATE 12,25 : PRINT "R" 'ADD TRADEMARK SIGN
250 CIRCLE (195,91),8
    
```

**GET
Statement (Graphics)**

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

**GOSUB...RETURN
Statements**

Purpose To branch to, and return from, a subroutine.

Format GOSUB <line number>

 .
 .
 .

 RETURN [<line number>]

 line number in the GOSUB statement is the first line of the subroutine. In the RETURN statement, the optional <line number> indicates a specific line to return to from the subroutine.

Comments The GOSUB and RETURN statements allow you to create subroutines that may be called any number of times from anywhere in your program. A subroutine may also be called from within another subroutine.

When the GOSUB statement is encountered, program control is transferred to the specified line number. Program control continues as normal from the specified line number until a RETURN statement is encountered. If the RETURN statement does not contain an optional line number, program control is transferred back to the statement following the most recently executed GOSUB statement. If the RETURN statement contains an optional line number, program control is transferred to the specified line number.

A subroutine may contain more than one RETURN statement if you need to return from different points in the subroutine.

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

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

**GOSUB and RETURN
Statements**

This example shows the general form of a subroutine. The GOSUB statement in line 10 calls, or transfers control to the subroutine at line 40. Program control is transferred to line 40 and statements are executed until the RETURN statement is encountered in line 70. The RETURN statement transfers program control back to the statement after the calling GOSUB, which in this example is line 20. The END statement in line number 30 prevents the program from entering the subroutine a second time.

Notes Care must be taken when using the RETURN with an optional line number. All GOSUB, WHILE, and FOR statements that were active at the time of the calling GOSUB remain active.

The ON...GOSUB statement can be used to branch to a subroutine based on the results of an expression.

**GOTO
Statement**

Purpose To branch unconditionally out of the normal program sequence to a specified line number.

Format GOTO <line number>

 line number is a valid line number in the program containing the next statement to be executed.

Comments When the GOTO statement is executed, program control is transferred to the specified line number. If <line number> is the line number of an executable statement, that statement and those following are executed. If <line number> refers to a non-executable statement (such as REM, DIM, or DATA), execution continues at the first executable statement encountered after the specified <line number>.

 If <line number> does not exist, an "Undefined line number" error message is displayed.

Example 10 INPUT "Enter radius (0 to end) - ",RADIUS
 20 IF RADIUS = 0 THEN END
 30 PRINT "Radius =" RADIUS,
 40 AREA = 3.14 * RADIUS^2
 50 PRINT "Area =" AREA
 60 PRINT
 70 GOTO 10

RUN
 Enter radius (0 to end) - 5
 Radius = 5 Area = 78.5

Enter radius (0 to end) - 7
 Radius = 7 Area = 153.86

Enter radius (0 to end) - 0
 Ok

The GOTO statement in line number 70 allows the program to repeat, or loop until a value of 0 is entered for RADIUS in the INPUT statement in line number 10.

Notes The GOTO statement can be used in the direct mode to enter a program at a specific line number. This can be helpful in debugging a program.

**HEX\$
Function**

Purpose To return a string that represents the hexadecimal value of the decimal argument.

Format HEX\$(x)

 x is a numeric expression in the range -32768 to 65535.

Comments The HEX\$ function evaluates the numeric expression x, rounds the result to an integer, and then returns the hexadecimal equivalent. If the resulting integer is negative, HEX\$ returns the value in the two's complement form.

Examples PRINT HEX\$(14)
 E
 Ok
 PRINT HEX\$(8.5)
 9
 Ok
 PRINT HEX\$(-1)
 FFFF
 Ok

**IF
Statement**

Purpose To make a decision regarding program flow based on the result returned by an expression.

Format IF <expression> THEN <clause> [ELSE <clause>]
 IF <expression> GOTO <line number> [ELSE <clause>]

expression is an expression composed of relational and/or logical operators which evaluates to a true (non-zero) or false (zero) value.

clause is a BASIC statement, a sequence of statements (separated by colons), or a program line number.

line number is a valid program line number.

Comments If the result of <expression> is true (non-zero), the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number.

If the result of <expression> is false (zero), the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. If not, execution continues with the next executable statement.

The IF...THEN...ELSE statement is just one statement. The ELSE portion of the statement cannot appear in the next program line. For example:

```
10 IF A = B THEN PRINT "TRUE" ELSE PRINT "FALSE"
```

is valid; whereas, the following is invalid:

```
10 IF A = B THEN PRINT "TRUE"
20 ELSE PRINT "FALSE"
```

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example:

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

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

**IF
Statement**

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

will not print "A<>C" when A<>B.

Examples 200 IF I THEN GET#1,I

This statement GETs record number I if I is not zero.

```
100 IF(I<10) OR (I>20) THEN FLAG = 1 : GOSUB 500
```

In this example, a test determines if I is in the range 10 to 20. If I is not in this range, FLAG is set to 1 and the subroutine at line number 500 is executed. If I is in this range, execution continues with the next line.

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

This statement results in A\$ being displayed on the screen or printed on the printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, A\$ is printed on the printer; otherwise, A\$ is displayed on the screen.

Notes

When using IF to test equality for a value that is the result of a single- or double-precision calculation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

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

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6 (0.000001).

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error message is displayed unless the specified line number exists in the program currently in memory.

**INKEY\$
Function**

Purpose To read a character from the keyboard.

Format INKEY\$

Comments The INKEY\$ function reads a single character from the keyboard buffer. The value returned will be in one of three forms:

- * a null string (length zero) indicating no characters are pending at the keyboard
- * a one-character string containing the actual character read from the keyboard
- * a two-character string indicating a special extended code. The first character returned will be the null character (ASCII code 000). The second character of the extended code is usually the scan code for the key pressed. For a listing of the extended codes refer to Appendix B.

The result of the INKEY\$ function must be assigned to a string variable before the character can be used in subsequent statements or functions.

The INKEY\$ function does not display the character on the screen and passes all characters to your program except for:

<Ctrl>/<Break>	Break function
<Ctrl>/<Num Lock>	Pause function
<Alt>/<Ctrl>/	System reset
<Ctrl>/<PrtSc>	Echo on printer
<Shift>/<PrtSc>	Print screen

Example 40 PRINT "Press C to Continue, A to Abort ";
 50 A\$ = ""
 60 WHILE A\$ <> "C" AND A\$ <> "A" : A\$ = INKEY\$: WEND
 70 PRINT A\$ 'ECHO RESPONSE
 80 IF A\$ = "A" GOTO 1000 'END PROGRAM

This program segment gives the user the choice of continuing with or aborting the program. Line number 60 reads the user's selection from the keyboard and accepts only an upper-case C or A as a valid answer. The WHILE...WEND loop continues looping until an A or C is pressed. Line 70 displays the user's choice, and line 80 branches based on that choice.

**INP
Function**

Purpose To return the byte read from a port.

Format INP(n)

n is an expression in the range -32768 to 65535
 representing a port to read from.

Comments INP is the complementary function to the OUT statement.

If n is negative, the INP function uses the value
(65536 + n) as the port to read.

INP performs the same function as the IN assembly
language instruction. Refer to the Technical Reference
for a description of valid port numbers (I/O
addresses).

Example 100 A = INP(513)

This statement reads a byte from port 513 and assigns
the value to variable A.

In 8086/8088 assembly language, this is equivalent to:

```
MOV DX,513  
IN  AL,DX
```

**INPUT
Statement**

Purpose To allow input from the keyboard during program execution.

Format INPUT[;] [<prompt>{;|,}]<variable>[,<variable>]...

prompt is a string constant used to prompt for the desired input.

variable is the name of a variable to accept the input. The variable can be numeric, string, or an array element.

Comments When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If a <prompt> and semicolon (;) are included, the prompt string is printed before the question mark. If a <prompt> and comma are included, just the prompt is displayed (the question mark is suppressed). The required data must be entered at the keyboard and the <Enter> key pressed to continue with program execution.

If the INPUT statement is immediately followed by a semicolon, the carriage return/line feed sequence is not produced when you press the <Enter> key at the end of the input. This means the cursor remains on the same line as your response.

The data that you enter is assigned to the variable(s) listed. The number of data items you enter must be the same as the number of variables in the list, and must be separated by commas.

The type of each data item that you enter must agree with the type specified by the variable name. Strings entered in response to an INPUT statement do not need to be surrounded by quotation marks unless they contain commas or significant leading or trailing spaces.

Responding to an INPUT statement with too many, too few, or the wrong type of value (string instead of numeric) results in the message "?Redo from start" to be displayed. You must then reenter the correct response. Assignment of input values is not made until an acceptable response is received.

**INPUT
Statement**

Examples 10 INPUT X
 20 PRINT X " SQUARED IS" X^2
 30 END
 RUN
 ? 5
 5 SQUARED IS 25
 Ok

In this example, the 5 was entered by the user in response to the question mark.

10 PI = 3.141593
20 INPUT "WHAT IS THE RADIUS";RADIUS
30 AREA = PI * RADIUS^2
40 PRINT "THE AREA OF THE CIRCLE IS" AREA
50 END
RUN
WHAT IS THE RADIUS? 7.4 (User enters 7.4)
THE AREA OF THE CIRCLE IS 172.0336
Ok

In this example, a prompt was added to the INPUT statement. When the INPUT statement is executed, the prompt and a question mark are displayed.

**INPUT#
Statement**

Purpose To read data items from a sequential device or file and assign them to program variables.

Format INPUT #<file number>,<variable>[,<variable>] ...

file number is the number under which the file was opened for input in the OPEN statement.

variable is the variable name to be assigned to the next data item from the file or device. The variable may be numeric, string, or an array element, but it must match the data type being assigned to it.

Comments The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and linefeeds are ignored. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a number. The number terminates on a space, carriage return, linefeed, or comma.

If GWBasic is scanning the sequential data file for a string item, it will also ignore leading spaces, carriage returns, and linefeeds. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string and terminates on a comma, carriage return, or linefeed (or after 255 characters have been read). If the end of the file has been reached when a numeric or string item is being INPUT, the item is cancelled and an "Input past end" error message is displayed.

The space between the T of INPUT and the # sign is optional.

Example 50 INPUT #2,A,B,C

This line would read the next three numbers from the file or device that was opened as #2 and assign them to the variables A, B, and C.

**INPUT\$
Function**

Purpose To return a string of characters from the keyboard or a file .

Format INPUT\$(n[, [#]<file number>])

n is the number of characters to be read.

file number is the number under which the file was opened in the OPEN statement.

Comments If a <file number> is not entered, the characters are read from the keyboard. The keyboard characters are not echoed on the screen. All characters, including control characters (except <Ctrl>/<Break> and <Ctrl>/<Alt>/), are passed to the INPUT\$ function. The <Ctrl>/<Break> is used to interrupt the execution of the INPUT\$ function. The characters are read from the keyboard as the keys are pressed and do not require the <Enter> key to be pressed to send them to the INPUT\$ function.

If a <file number> is entered, the INPUT\$ function reads in the next n characters in the file. The INPUT\$ function does not read in data items, but reads the actual characters in the file; this includes delimiting spaces, carriage return characters, and line feed characters. The INPUT\$ function will not read the ^Z end-of-file character; an "Input past end" error message is displayed if attempted.

Examples 10 OPEN "I",1,"DATA"
 20 WHILE NOT EOF(1)
 30 PRINT ASC(INPUT\$(1,#1));
 40 WEND
 50 CLOSE

This example opens the sequential file DATA and displays the ASCII code for each character in the file.

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP ";
110 X$=INPUT$(1) : PRINT X$ : PRINT
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
```

This example reads a single character from the keyboard in response to the statement displayed by the PRINT statement in line 100. If a P or S is not input, the program loops back to line number 100.

**INSTR
Function**

Purpose To search for the first occurrence of a string within another string and return the position at which the match is found.

Format INSTR([n],[x\$],y\$)

n is a numerical expression in the range 1 to 255 representing an optional offset into the searched string to be used as the starting point for the search.

x\$ is a string variable, string expression, or string constant to be used as the searched string.

y\$ is a string variable, string expression, or string constant to be used as the search string.

Comments The INSTR function searches string x\$ for the first occurrence of string y\$ and returns its position in x\$. If an optional offset is included, the search starts at that character position in string x\$.

If offset n is greater than the number of characters in x\$ (LEN(x\$)), x\$ is null, or y\$ is not found in x\$, INSTR returns 0. If y\$ is null, INSTR returns n (or 1 if n is not specified).

Example 10 X\$ = "ABCDEB" : Y\$ = "B"
20 OFFSET = 1 : POSITION = 1
30 WHILE POSITION <> 0
40 POSITION = INSTR(OFFSET,X\$,Y\$)
50 IF POSITION <> 0 THEN PRINT POSITION :
 OFFSET = POSITION + 1
60 WEND
70 END
RUN
 2
 6
Ok

This example searches for string "B" within string "ABCDEB". The WHILE...WEND loop is used to search for all occurrences of string "B". Line number 20 sets the initial offset into the searched string as 1, or the beginning of the string. If "B" is found, line number 50 displays its location and then sets the offset to the next character position in X\$. The loop is executed again looking for the next occurrence of "B". The search is ended if INSTR returns a value of zero, indicating Y\$ was not found or the offset is now larger than the length of the searched string.

**INT
Function**

Purpose To return the largest integer less than or equal to the
 specified number.

Format INT(x)

 x is any numeric expression.

Comments The INT function evaluates the specified numeric
 expression and returns the largest integer that is less
 than or equal to that value.

Examples PRINT INT(99.89)
 99

 PRINT INT(-12.11)
 -13

Notes See the CINT and FIX functions for alternative methods
 of returning integer values.

**KEY
Statement (Key Trapping)**

Purpose To define six key traps in addition to the function keys and cursor movement keys.

Format KEY n,CHR\$(*<shift code>*)+CHR\$(*<scan code>*)

n is a numeric expression in the range 15 to 20 to be assigned to the key to be trapped.

shift code is a mask used to indicate the shifted state of the key being trapped. The codes for the shifted states are:

	Hex	Decimal
Shift	&H01, &H02, &H03	1, 2, 3
Ctrl	&H04	4
Alt	&H08	8
Num Lock	&H20	32
Caps Lock	&H40	64

Key trapping treats the left and right shift keys as the same; therefore, you can use a value of 1, 2, or 3 (the sum of 1 and 2). To trap a non-shifted key state, use a shift code value of zero (0).

scan code is a numeric expression in the range 1 to 83 representing the scan code for the key to be trapped. Refer to Appendix C for a keyboard diagram with scan codes.

Comments The KEY statement allows you to define key trapping sequences for six keys, or key sequences, in addition to the function keys and cursor movement keys defined in the ON KEY statement. This allows you to trap any single key on the keyboard, or any of the Ctrl, Shift, or Alt shifted key sequences. Multiple shift states, such as <Ctrl>/<Alt>/..., can be trapped by using a shift code equal to the sum of the individual shift codes. Therefore, for a <Ctrl>/<Alt>/... shift sequence, a shift code of 12 (&HC) would be used.

**KEY
Statement (Key Trapping)**

Key traps are processed in the following order:

1. The <Ctrl>/<PrtSc> key sequence, which activates the printer echo feature. Defining the <Ctrl>/<PrtSc> sequence as a user defined key trap does not disable its function as a switch for the print echo feature.
2. The function keys F1 to F10, Cursor Up, Cursor Left, Cursor Right, and Cursor Down keys (1-14). Because these keys are predefined as keys 1 through 14, defining scan codes 59-68, 72, 75, 77, or 80 as a user defined key trap has no effect.
3. Keys 15 to 20 that you define using the KEY statement.

```

Example 10 KEY 15,CHR$(0)+CHR$(30) 'LOWER-CASE A
        20 KEY 16,CHR$(64)+CHR$(30) 'UPPER-CASE A
        30 KEY 17,CHR$(1)+CHR$(30) 'SHIFT A
        40 KEY 18,CHR$(4)+CHR$(30) 'CONTROL A
        50 KEY 19,CHR$(8)+CHR$(30) 'ALT A
        60 KEY 20,CHR$(12)+CHR$(30) 'CTRL ALT A
        70 FOR I = 15 TO 20
        80 ON KEY(I) GOSUB 1000
        90 KEY(I) ON
        100 NEXT I
    
```

This example shows the statements used to trap six different states of the A key. Line number 60 uses a shift code of 12 (4 + 8) to indicate the <Ctrl>/<Alt>/state is being trapped. The FOR...NEXT loop in line 70 through 100 is used to assign the same subroutine for all key traps (line 80) and to turn each key trap on (line 90).

Notes Trapped keys are not placed in the keyboard buffer to be read by BASIC.

You can prevent a user from breaking out of your program by trapping the <Ctrl>/<Break> and <Ctrl>/<Alt>/ key sequences.

KEY

Statement (Soft Keys)

Purpose To assign soft key values to the function keys and display the values.

Format KEY n,x\$
 KEY LIST
 KEY ON
 KEY OFF

n is a function key number in the range 1 to 10.

x\$ is a string expression to be assigned to the specified function key.

Comments The KEY statement allows the function keys to be used as soft keys. A soft key lets you enter a sequence of characters by simply pressing that key. A string of up to 15 characters may be assigned to each of the ten function keys.

KEY n,x\$ assigns the string x\$ to function key Fn, where n is in the range 1 to 10. x\$ may be up to 15 characters in length. If x\$ is longer than 15 characters, only the first 15 are assigned. Assigning the null string (length zero) to a function key disables it as a soft key. If a value for n is entered outside the range 1 to 10, an "Illegal function call" error message is displayed.

Initially, the soft keys are assigned the following values:

F1 - LIST_	F6 - ,"LPT1:"<CR>
F2 - RUN<CR>	F7 - TRON<CR>
F3 - LOAD"	F8 - TROFF<CR>
F4 - SAVE"	F9 - KEY_
F5 - CONT<CR>	F10- SCREEN 0,0,0<CR>

<CR> indicates ASCII character 13, a carriage return
 _ indicates a blank space

Once soft keys have been designated, they can be displayed with the KEY ON, KEY OFF, and KEY LIST statements.

KEY ON causes the soft key values to be displayed on the 25th line on the screen. When the screen width is 40 characters, the first five soft keys are displayed; when the width is 80, all ten soft keys are displayed. In either screen width, only the first 6 characters assigned to each key are displayed. ON is the default state for the soft key display.

KEY**Statement (Soft Keys)**

If the carriage return character, CHR\$(13), is included in the string, it is displayed as a left arrow on the screen.

KEY OFF clears the soft key display from the 25th line, making that line available for program use. It does not disable the soft keys.

KEY LIST displays all ten soft key values on the screen. All 15 characters of each soft key are displayed.

```
Examples 10 KEY OFF 'CLEARS THE 25TH DISPLAY LINE
          20 KEY 1,"PRINT "
          30 KEY 2,"WHILE "
          40 KEY 3,"WEND"
          50 KEY 4,"INPUT "
          60 KEY 5,"OPEN "
          70 KEY 6,"INPUT#"
          80 KEY 7,"PRINT#"
          90 KEY 8,"DRAW "
          100 KEY 9,"PAINT "
          110 KEY 10,"RUN"+CHR$(13)
          120 KEY ON
```

To aid in program entry, the above program could be used to reassign the soft keys to some of the more commonly used GWBASIC statements. The CHR\$(13) used in the assignment of function key 10 produces the same result as pressing the <Enter> key.

```
20 FOR I = 1 TO 10 : KEY I,"" : NEXT I
```

The above line would disable all ten soft keys.

Notes

When a soft key is assigned, the INKEY\$ function returns one character of the soft key string each time it is called. If the soft key is disabled, the INKEY\$ function will return the two character extended code for that function key (refer to extended codes in Appendix B).

**KEY(n)
Statement**

Purpose To enable or disable key trapping.

Format KEY(n) ON
KEY(n) OFF
KEY(n) STOP

n is a numeric expression in the range 1 to 20 representing the key to be trapped.

1-10	function keys F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-20	user defined (see KEY statement for key trapping)

Comments The KEY(n) ON statement enables the key trapping defined by the ON KEY(n) statement. While trapping is enabled, and if a non-zero line number is specified in the ON KEY(n) statement, BASIC checks between execution of each statement to see if the specified key has been pressed. If it has, the GOSUB portion of the ON KEY(n) statement is executed.

KEY(n) OFF disables the key trapping. If the key is pressed, it is not remembered.

KEY(n) STOP disables the key trapping. If the key is pressed, the event is remembered and the GOSUB portion of the ON KEY statement is executed as soon as a KEY(n) ON statement is executed.

Example 10 KEY 1,"QUIT"
20 KEY ON 'TURN ON SOFT KEYS
30 ON KEY(1) GOSUB 1000 'END PROGRAM
40 KEY(1) ON

This program segment sets up function key F1 to be used as an easy way to terminate the program. Line 10 assigns the string "QUIT" to softkey F1. Line 20 displays the soft keys on line 25 of the display screen. Line 30 defines the key trap for function key F1, sending it to a routine at line 1000. Line 40 enables the key trapping for function key F1.

Notes For additional information on key event trapping, refer to the ON KEY Statement.

**KILL
Command**

Purpose To delete a file from disk.

Format KILL <filespec>

 filespec is a valid file name.

Comments The KILL command can be used to delete all types of disk files. If the file to be deleted has a filename extension, the extension must be included in the KILL command. The entered filespec may contain the question mark (?) or asterisk (*) wildcard characters to delete multiple files.

Examples KILL "DATAFILE.DAT"

 10 X\$ = "DATAFILE.DAT"
 20 KILL X\$

Both of the above examples could be used to delete the file DATAFILE.DAT from the default drive.

KILL "B:SALES\SALES84\SOFTWARE.DAT"

The above example deletes the data file SOFTWARE.DAT in the SALES84 subdirectory on drive B.

Notes If a KILL command is given for a file that is currently OPEN, a "File already open" error message is displayed.

 If the filespec in the KILL command does not exist, a "File not found" error message is displayed.

 The KILL command cannot be used to delete a sub-directory.

LEFT\$

Function

Purpose To return a string comprised of the leftmost n characters of string x\$.

Format LEFT\$(x\$,n)

x\$ is any string expression.

n is a numeric expression in the range 0 to 255.

Comments The LEFT\$ function is used to return a string from the left n characters of string x\$.

If n is greater than the number of characters in x\$ (LEN(x\$)), the entire string (x\$) is returned. If n is zero, the null string (length zero) is returned.

Example

```
10 A$ = "BASIC PROGRAMMING"
20 PRINT LEFT$(A$,5)
RUN
BASIC
Ok
```

Notes The MID\$ and RIGHT\$ functions can also be used to return portions of a string.

**LEN
Function**

Purpose To return the number of characters in a string expression.

Format LEN(x\$)
x\$ is any string expression.

Comments The LEN function returns the number of characters in the specified string expression. Unprintable characters and blanks are counted as characters.

Example 10 LOCATION\$ = "SUNNYVALE, CA"
20 PRINT LEN(LOCATION\$)
RUN
13
Ok

**LET
Statement**

Purpose To assign the value of an expression to a variable.

Format [LET] <variable> = <expression>

variable is a valid variable name used to receive the value of <expression>. The variable may be string, numeric, or an array element.

expression is an expression whose value is assigned to <variable>. The value of <expression> must be of the same type (string or numeric) as <variable>.

Comments The word LET is optional. The equal sign is sufficient for assigning an expression to a variable name.

Examples	110 LET D = 12	110 D = 12
	120 LET E(1) = 12^2	120 E(1) = 12^2
	130 LET F\$ = "HELLO"	130 F\$ = "HELLO"
	140 LET SUM = D + E(1)	140 SUM = D + E(1)

The above examples show two ways of assigning values to the variables D, E(1), F\$, and SUM.

LINE
Statement

Purpose To draw a line or box on the screen in the graphics mode.

Format LINE [(x1,y1)]-(x2,y2)[,[<color>]][,[B[F]]][,<style>]]

(x1,y1) is the coordinate for the starting point of the line. If omitted, the current graphics point (last point referenced) is used.

(x2,y2) is the ending point for the line.

color is the number of the color in which the line should be drawn. In medium resolution, color is in the range 0 to 3. 0 indicates the background color, colors 1 to 3 indicate the colors of the current palette (see the COLOR statement). The default color is the foreground color, color 3. In high resolution, color can be 0 or 1. 0 (zero) indicates black, 1 indicates white (the default color). If <color> is not within the above ranges, an "Illegal function call" error message is displayed.

,B option draws a rectangle using the two indicated points as opposite corners. This is equivalent to the following four LINE statements:

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

,BF options draws the same box as the B option, but also fills the box using the line color.

style is an expression representing a 16-bit mask used to change the line style. The <style> option can be used for lines and boxes (B option), but cannot be used for filled boxes (BF option). If <style> is used with the filled box (BF) option, a "Syntax error" error message is displayed. <style> must be in the range -&HFFFF to &HFFFF, or -32768 to 32767 or an "Overflow" error message is displayed. The default value for <style> is hex FFFF.

**LINE
Statement**

Comments The LINE command is used in the graphics mode to draw straight lines or rectangles on the screen. The points for the line can be given in either absolute or relative form (refer to Chapter 4 for information on specifying screen coordinates).

If lines or rectangles are specified that contain out-of-range points, line clipping occurs and only those points with valid screen coordinates are shown.

The line <style> option can be used to draw dotted or dashed lines. The LINE statement uses the bit pattern of the <style> entry to plot the points on the screen. It rotates through the bit pattern, displaying points in the specified color if the bit is 1 (one), skipping over the point (leaving the current state) if the bit is 0 (zero). To draw a dotted line using every other point, a <style> entry of hex AAAA would be used. This entry would provide a bit pattern of:

1010101010101010

Examples The following examples assume you are in the medium resolution graphics mode.

40 LINE -STEP(10,0)

Draws a line 10 points to the right from the last point referenced. The line is drawn in color 3 of the current palette.

20 LINE (0,0)-(319,199)

Draws a diagonal line across the screen from point 0,0 to point 319,199.

30 LINE (0,100)-(319,100),2,,&H0F0F

Draws a horizontal dashed line across the middle of the screen in color 2 of the current palette.

40 LINE (10,10)-(20,20),,BF

Draws a filled rectangle in color 3 of the current palette with corner coordinates of (10,10), (10,20), (20,20), and (20,10)

Notes The last point referenced after the LINE statement is point (x2,y2).

**LINE INPUT
Statement**

Purpose To input an entire line (up to 255 characters) from the keyboard into a string variable, ignoring delimiters.

Format LINE INPUT[;][<prompt>;]<variable>

prompt is a string constant that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of prompt string.

variable is the name of a valid string variable or string array element used to accept the line input. All characters entered up to the terminating <Enter> are assigned to the <string variable>, except trailing blanks.

Comments The LINE INPUT statement allows you to assign an entire line of input from the keyboard to a string variable. The input line can contain commas, semicolons, quotation marks, and other forms of punctuation normally used as delimiters.

If a linefeed key sequence (<Ctrl>/<Enter>) is entered, the cursor is moved to the beginning of the next line, the linefeed character (hex 0A) is placed in the string, and input continues.

If LINE INPUT is immediately followed by a semicolon, the <Enter> used to end the input line does not echo a carriage return/linefeed sequence on the screen. The cursor is left on the same line as the your input.

If <Ctrl>/<Break> is entered, the LINE INPUT statement is aborted and GWBASIC returns to the command level. You may enter CONT to resume execution at the LINE INPUT statement.

Example 10 LINE INPUT "ENTER SOMETHING ";A\$
 20 PRINT : PRINT A\$
 RUN
 ENTER SOMETHING **THIS "EXAMPLE" SHOWS SOME OF THE**
 THINGS THE LINE INPUT CAN DO, RIGHT?

 THIS "EXAMPLE" SHOWS SOME OF THE
 THINGS THE LINE INPUT CAN DO, RIGHT?
 Ok

The bold face shows what the user entered. A linefeed was entered after the word "THE" to move the text to the next line.

LINE INPUT#

Statement

Purpose To read an entire line (up to 255 characters), ignoring delimiters, from a sequential disk data file into a string variable.

Format LINE INPUT #<file number>,<variable>

file number is the number under which the file was opened in the OPEN statement.

variable is the name of a valid string variable or string array element used to accept the line input.

Comments LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/linefeed character sequence and the next LINE INPUT# reads all characters up to the next carriage return. If a linefeed character is encountered, it is returned as part of the string.

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII format is being read as data by another program.

Example 10 OPEN "O",1,"NAMES"
 20 PRINT #1,"TELEVIDEO 1170 MORSE SUNNYVALE, CA 94086"
 30 CLOSE
 40 PRINT "NAME ADDRESS" : PRINT
 50 OPEN "I",1,"NAMES"
 60 LINE INPUT #1, C\$
 70 PRINT C\$
 80 CLOSE
 RUN
 NAME ADDRESS

TELEVIDEO 1170 MORSE SUNNYVALE, CA 94086
 Ok

This example places the TeleVideo name and address in the sequential file NAMES. The LINE INPUT# statement is used to read the name and address back out of the file.

**LIST
Command**

Purpose To list all or part of the program currently in memory on the screen or optionally on a specified device.

Format LIST [<start line>][-<end line>][,<device>]

start line is a program line number in the range 0 to 65529 indicating the first line number to be listed. A period (.) can be used to indicate the current line.

end line is a program line number in the range 0 to 65529 indicating the last line number to be listed. A period (.) can be used to indicate the current line.

device is a string expression representing a file specification or output device. If <device> is omitted, the listing is displayed on the screen.

Comments GWBASIC always returns to command level after a LIST is executed.

If the LIST command is entered without any options, the entire program currently in memory is listed on the screen. The <Ctrl>/<Num Lock> key sequence can be used to stop or pause the listing, allowing you to look at the lines on the screen. Press any key to continue with the listing. To interrupt the listing and return to the command level, press the <Ctrl>/<Break> key sequence.

If a <start line> and <end line> are entered, all the program lines in the range <start line> to <end line> inclusive are listed. If only the <start line> is specified, only that line number is listed.

If a <start line> and a hyphen (-) are entered, all the program lines from <start line> to the end of the program are listed.

If a hyphen (-) and an <end line> are entered, all the program lines from the beginning of the program up to the <end line> are listed.

When an optional <device> is entered, the listing is output to the device or file specification entered. When you list to a disk file, the program lines are stored in the ASCII format. This allows the program lines to be used by the MERGE command.

LIST Command

Examples LIST

Lists the entire program currently in memory.

LIST 500

Lists line 500.

LIST 150-

Lists all lines from 150 to the end of the program.

LIST -1000

Lists all lines from the beginning of the program through line 1000.

LIST 150-1000

Lists lines 150 through 1000, inclusive.

LIST 150-1000,"LPT1:"

Lists lines 150 through 1000 on the printer.

LIST ,"B:EXTRA"

Lists the current program to a file named EXTRA on drive B. The program is stored in the ASCII format.

**LLIST
Command**

Purpose To list all or part of the program currently in memory on the printer (LPT1:).

Format LLIST [<start line>][-<end line>]

start line is a program line number in the range 0 to 65529 indicating the first line number to be listed. A period (.) can be used to indicate the current line.

end line is a program line number in the range 0 to 65529 indicating the last line number to be listed. A period (.) can be used to indicate the current line.

Comments The LLIST command allows you to list the program currently in memory on the printer. The <start line> and <end line> options work the same as they do in the LIST command.

GWBasic always returns to the command level after a LLIST is executed.

Examples LLIST

Lists the entire program currently in memory.

LLIST 500

Lists line 500.

LLIST 150-

Lists all lines from 150 to the end of the program.

LLIST -1000

Lists all lines from the beginning of the program through line 1000.

LLIST 150-1000

Lists lines 150 through 1000, inclusive.

**LOAD
Command**

Purpose To load a file from disk into memory and optionally
 run it.

Format LOAD <filespec>[,R]

filespec is the name of a disk file to be loaded into
 memory. <filespec> may consist of an
 optional drive name, an optional path, a
 filename, and an optional filename extension.

Comments The <filespec> entered is the filename that was used
 when the file was saved with the SAVE command. If a
 filename extension is not included, GWBasic assumes
 an extension of .BAS. GWBasic returns to the command
 level after the program is loaded.

If the R option is added, GWBasic automatically runs
the program after it has been loaded.

LOAD closes all open files and deletes all variables
and program lines currently residing in memory before
it loads the designated program. However, if the R
option is added, all open data files remain open.
Thus, LOAD with the R option may be used to chain
several programs (or segments of the same program).
Information may be passed between the programs using
their disk data files.

Examples LOAD "STARTREK",R

Loads and runs the program STARTREK.BAS from the
default drive.

LOAD "B:MYPROG"

Loads the program MYPROG.BAS from drive B, but does not
run the program.

Notes The command LOAD <filespec>,R is equivalent to the
 command RUN <filespec>.

**LOC
Function**

Purpose To return the current position in a file. In a random access file, LOC returns the number of the last record read or written. In a sequential file, LOC returns the number of 128-byte records read from or written to the file since it was opened.

Format LOC(<file number>)

 file number is the number under which the file was opened in the OPEN statement.

Comments When a file is opened for sequential input, GWBasic reads the first sector of the file, so LOC will return a 1 even before any input from the file occurs.

 For a communications file, LOC returns the number of characters in the input buffer waiting to be read. If there are more than 255 characters in the buffer, LOC returns 255.

Example 200 IF LOC(1) > 50 THEN END

 In this example, the program will end after we've gone beyond the 50th record in file number 1.

**LOCATE
Statement**

Purpose To move the cursor to the specified position. Optional parameters determine the shape of the cursor and turn the cursor on or off.

Format LOCATE [<row>][, [<col>]][, [<cursor>]][, <start>[, <stop>]]]

row is a numeric expression in the range 1 to 25 indicating a line number on the screen.

col is a numeric expression in the range 1 to 80 (1 to 40 in the 40 column mode) indicating a column number (character position) on the screen.

cursor is a value indicating whether the cursor should be visible or not while a program is running. A 0 (zero) turns the cursor off so it is not visible, a 1 (one) turns it on, making it visible.

start is a numeric expression in the range 0 to 31 indicating the starting scan line of the cursor.

stop is a numeric expression in the range 0 to 31 indicating the ending scan line of the cursor.

<cursor>, <start>, and <stop> do not apply to the graphics mode.

Comments The LOCATE statement with the <row> and <col> options is used to move the cursor to a specified position on the screen. This allows your program to print anywhere on the screen.

The LOCATE statement also allows you to display characters on line 25. When the soft keys have been turned off with the KEY OFF statement, a LOCATE 25,<col> and PRINT... statement can be used to display a messages on line 25. The screen will scroll up normally from line 24, but if the screen is cleared with CLS or PRINT CHR\$(12), line 25 is also cleared.

Note that the LOCATE statement uses the <row>,<col> format, with the vertical coordinate first, whereas the graphics commands use a (x,y) format, with the horizontal coordinate first.

**LOCATE
Statement**

The LOCATE statement with the <cursor> option can be used to turn the cursor on (visible) or off (invisible). When GWBASIC is running your program, the cursor is normally off. The LOCATE ,,1 statement could be used to display the cursor while your program is running.

The <start> and <stop> options allow you to change the shape of the cursor. The cursor shape is determined by the number of scan lines that are turned on. The top scan line is 0 (zero), and the bottom scan line is 8. To display a full rectangle cursor, you would use the LOCATE ,,0,8 statement. If the <start> scan line is larger than the <stop> scan line, you will get a two part cursor. The cursor wraps around from the bottom to the top. Zero to 31 are valid <start> and <stop> values, but only values 0 to 8 affect the visible shape of the cursor.

If the <stop> option is omitted, the <stop> value is set equal to the <start> value. The default underline cursor would be set using the LOCATE ,,7 statement.

If a parameter is omitted from the LOCATE statement, it retains its current value.

If a parameter value is entered outside of the specified range, an "Illegal function call" error message is displayed. In this case, the previous values are retained.

Examples 10 LOCATE 1,1

Moves the cursor to the home position in the upper-left corner of the screen.

20 LOCATE ,,1

Makes the cursor visible; its position remains unchanged.

30 LOCATE ,,7

This LOCATE statement only changes the shape of the cursor. It sets the cursor to display a single scan line at the bottom of the character box, starting and ending on scan line 7 (this is the default underline cursor in the 80-column text mode).

**LOF
Function**

Purpose To return the length of a file in bytes.

Format LOF(<file number>)

file number is the number under which the file was opened in the OPEN statement.

Comments The LOF function returns the number of bytes allocated to the specified file.

For communications, LOF returns the number of free bytes in the input buffer. The number of characters in the buffer can be determined by subtracting the value of LOF from the input buffer size (256 bytes by default).

Example 110 IF REC*RECSIZE > LOF(1) THEN PRINT "INVALID ENTRY"

In this example, the variables REC and RECSIZE contain the record number and record length, respectively. The calculation determines whether the specified record is beyond the end-of-file.

**LOG
Function**

Purpose To return the natural logarithm of a specified number.

Format LOG(x)

 x is a numeric expression greater than 0.

Comments The LOG function returns the natural (base e) logarithm of expression x. If x is not greater than zero, an "Illegal function call" error message is displayed.

Example PRINT LOG(EXP(2))
 2
 Ok

**LPOS
Function**

Purpose To return the current position of the printer's print head within the printer buffer.

Format LPOS(x)

x is numeric expression indicating which printer is being tested. Printers are indicated as follows:

0 or 1	LPT1:
2	LPT2:
3	LPT3:

Comments LPOS does not necessarily give the physical position of the print head.

Example 100 IF LPOS(0) > 60 THEN LPRINT CHR\$(13)

In this example, if the length of the line is greater than 60 characters, a carriage return is sent to the printer to advance to the next line.

**LPRINT and LPRINT USING
Statements**

Purpose To print data on the printer.

Format LPRINT [<expression>][;][<expression>] ...

LPRINT USING <string exp>;<expression>;
[<expression>] ...

expression is a numeric or sting expression to be printed. <expression>s may be separated by commas or semicolons.

string exp is a string expression identifying the format to be used for printing the following expressions. (Refer to the PRINT USING statement for a description of formats.)

Comments The LPRINT and LPRINT USING statements function like the PRINT and PRINT USING statements, except the output goes to the printer.

LPRINT assumes an 80-character wide printer. This means GWBasic automatically inserts a carriage return/line feed after printing 80 characters. If you need to print exactly 80 characters, end the LPRINT statement in a semicolon, otherwise two carriage return/line feed sequences are executed. The line length can be changed using a WIDTH "LPT1:" statement.

Examples

```

240 GOSUB 320
250 FOR I = START TO FINISH
260 LPRINT NAME$(I) TAB(25) ADDR$(I) TAB(65) CITY$(I)
270 IF I/55 = INT(I/55) THEN GOSUB 310
280 NEXT I
290 RETURN
300 '***** HEADER *****
310 LPRINT CHR$(12)
320 LPRINT "NAME" TAB(25) "ADDRESS" TAB(65) "CITY"
330 LPRINT 'PRINT A BLANK LINE
340 RETURN
    
```

This program subroutine could be used to print a listing of the names and address stored in the string arrays NAMES\$, ADDR\$, and CITY\$. The subroutine in lines 310 to 340 is used to bring the paper to the top of the form and to print the header and a blank line at the top of the page. The FOR...NEXT loop in lines 250 to 280 prints the names and addresses. Line 270 is used to limit the listing to 55 names per page. After 55 names are printed, the HEADER subroutine is called to advance the paper to the next page.

**LPRINT and LPRINT USING
Statements**

Notes A "Device Timeout" error message indicates the printer did not respond within a predetermined amount of time when GWBasic is trying to write to the printer. This could occur if you use an LPRINT statement when the printer is not connected.

**LSET and RSET
Statements**

Purpose To move data into a random file buffer (in preparation for a PUT statement).

Format LSET <string variable>=<string expression>
 RSET <string variable>=<string expression>

 string variable is the name of a string variable that was used to define a record field in a random file buffer with the FIELD statement.

 string expression is a string expression containing the information to be placed into the random file buffer in the field identified by <string variable>.

Comments The LSET and RSET statements place data into the random file buffer so it can be written from the buffer to the file by the PUT statement.

 If <string expression> requires fewer bytes than were specified for the <string variable> in the FIELD statement, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right.

 Numeric values must be converted to strings using the MKI\$, MKS\$, or MKD\$ functions before they can be LSET or RSET.

Examples 150 LSET AMOUNT\$ = MKS\$(AMOUNT)

 This example converts the numeric variable AMOUNT to a 4-byte string, and then left-justifies it in the buffer field AMOUNT\$ in preparation for a PUT statement.

Notes LSET or RSET may also be used to left-justify or right-justify a string in a given field, even though the <string variable> was not defined in a FIELD statement. For example, the program lines:

```
110 A$ = SPACE$(20)
120 RSET A$ = N$
```

 right-justify the string N\$ in a 20-character field. This can be useful for formatting printed output.

**MERGE
Command**

Purpose To merge a specified disk file into the program
 currently in memory.

Format MERGE <filespec>

filespec is a string expression indicating the name of
 the file to be merged into the current
 program. The file must have been saved in
 the ASCII format (using the ,A option with
 the SAVE command) or a "Bad file mode" error
 message is displayed.

Comments The MERGE command searches for the specified file, and
 if found, merges the program lines from the disk file
 with the program lines currently in memory. If any
 lines in the disk file have the same line numbers as
 lines in the program in memory, the lines from the file
 on disk replace the corresponding lines in memory.

If <filespec> does not include a drive designation, the
TeleDOS default drive is assumed. If <filespec> does
not include a filename extension, GWBasic assumes a
filename extension of .BAS.

GWBasic always returns to the command level after
executing a MERGE command.

Example MERGE "B:NUMBERS"

This command merges the file named NUMBERS.BAS on drive
B into the program currently in memory.

**MID\$
Function**

Purpose To return a specified portion of a string.

Format MID\$(x\$,n[,m])

x\$ is any valid string expression.

n is a numeric expression in the range 0 to 255 indicating the starting character position in x\$. (MID\$ rounds n to an integer to determine the starting character position.)

m is a numeric expression in the range 0 to 255 indicating the number of characters to be returned from x\$. (MID\$ rounds m to an integer to determine the number of characters.) If m is omitted, or less than m characters remain in x\$ from position n, all the remaining characters are returned.

Comments The MID\$ function returns m characters from string x\$, starting at character position n. If n is greater than the number of characters in x\$ (LEN(x\$)), MID\$ returns a null string.

Example 10 A\$ = "GOOD "
 20 B\$ = "MORNING EVENING AFTERNOON"
 30 PRINT A\$ MID\$(B\$,9,7)
 RUN
 GOOD EVENING
 Ok

This example uses the MID\$ function to select the center word from string B\$.

Notes Also see the LEFT\$ and RIGHT\$ functions for alternative ways of returning portions of a string.

MID\$
Statement

Purpose To replace a portion of one string with another string.

Format MID\$(x\$,n[,m]) = y\$

x\$ is a string variable or string array element that will have its characters replaced.

n is a numeric expression in the range 0 to 255 indicating the character position where the replacement will start. (MID\$ rounds n to an integer to determine the starting position.)

m is a numeric expression in the range 0 to 255 indicating the number of characters in y\$ that will be used in the replacement. (MID\$ rounds m to an integer to determine the number of characters.) If m is omitted, the entire string y\$ is used in the replacement.

y\$ is a string expression containing the characters to be used in the replacement.

Comments The MID\$ statement replaces characters in string x\$, beginning at position n, with characters in string y\$. The option m indicates the number of characters from y\$ that will be used in the replacement.

The length of string x\$ does not change, regardless of whether a value for m was entered or not.

If a value for n or m is entered out of the specified range, an "Illegal function call" error message is displayed.

Example 10 CITY\$ = "KANSAS CITY, KANSAS"
 20 MID\$(CITY\$,14) = "MISSOURI"
 30 PRINT CITY\$
 RUN
 KANSAS CITY, MISSOU
 Ok

In this example, the MID\$ statement is used to change the state from Kansas to Missouri, but because the length of CITY\$ cannot change, only the first six letters of the state name Missouri were used.

**MKDIR
Command**

Purpose To create a directory on the specified disk.

Format MKDIR <path>

 path is a string expression of up to 128
 characters indicating the name of the new
 directory.

Comments The MKDIR command works like the TeleDOS MKDIR command.

Example MKDIR "B:SALES"

This command creates the sub-directory SALES on drive B. SALES is a sub-directory of the current directory on drive B. To create the sub-directory JOHN in directory SALES, the following command would be given:

MKDIR "B:SALES\JOHN"

The sub-directory JOHN could also be made using the following commands:

CHDIR "B:SALES"

MKDIR "B:JOHN"

**MKI\$, MKS\$, MKD\$
Functions**

Purpose To convert numeric values to string values.

Format MKI\$(<integer expression>)
 MKS\$(<single-precision expression>)
 MKD\$(<double-precision expression>)

Comments Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI\$ converts an integer to a 2-byte string. MKS\$ converts a single-precision number to a 4-byte string. MKD\$ converts a double-precision number to an 8-byte string.

These functions do not really change the bytes of data, but just the way GWBASIC interprets the bytes.

Example 100 FIELD #1,20 AS A1\$,4 AS A2\$
 110 LSET A1\$ = ITEM\$
 120 LSET A2\$ = MKS\$(AMOUNT)
 130 PUT #1

This example uses a random file with field variables defined by the FIELD statement in line 100. Field variable A1\$ holds an item description of up to 20 characters, and field variable A2\$ is intended to hold a single-precision numeric value. Line 120 uses the MKS\$ function to convert the single-precision variable AMOUNT to a 4-byte string. The LSET statement places the string representation into the file buffer. Line 130 actually writes the data from the buffer to the file.

Notes Refer to the CVI, CVS, and CVD functions to convert the string representation back to a numeric value.

NAME
Statement

Purpose To change the name of a disk file. The NAME statement performs the same function as the TeledOS RENAME command.

Format NAME <old filespec> AS <new filespec>

old filespec is the current filename. <old filespec> may include an optional device designation, an optional path, a filename, and an optional filename extension. If <old filespec> does not exist, a "File not found" error message is displayed.

new filespec is the new filename. <new filespec> must be a valid filename as described in Chapter 4. If <new filespec> already exists, a "File already exists" error message is displayed.

Comments The NAME command does not move the file. The file will exist on the same drive, in the same area on the disk, with the new filename.

If you try to rename a file with a new drive designation, a "Rename across disks" error message is displayed.

If <old filespec> contains a path, and <new filespec> does not, <new filespec> will be listed under the current directory.

If <new filespec> does not include a filename extension, the NAME command **does not** add a default extension.

Example NAME "ACCTS.BAS" AS "LEDGER"

In this example, file ACCTS.BAS on the default drive is renamed to LEDGER.

**NEW
Command**

Purpose To delete the program currently in memory and clear all variables.

Format NEW

Comments The NEW command is usually entered in the direct mode to clear memory before entering a new program. GWBASIC always returns to the command level after the NEW command is executed.

NEW closes all files and turns the trace off if it is on (see the TRON and TROFF commands).

Example NEW
 Ok

The program that was currently in memory has been deleted.

**OCT\$
Function**

Purpose To return a string that represents the octal value of
 a specified decimal argument.

Format OCT\$(x)

 x is a numeric expression in the range -32768 to
 65535.

Comments The OCT\$ function rounds expression x to an integer and
 returns its octal equivalent. If x is negative, the
 two's complement form is used.

Example PRINT OCT\$(24.3)
 30
 Ok

OCT\$ rounds 24.3 to 24, and then returns the octal
representation of 24, which is 30.

**ON COM(n)
Statement**

Purpose To specify the first line number of a subroutine to be performed when activity occurs on a communications channel.

Format ON COM(n) GOSUB <line number>

 n is the number of the communications channel (1 or 2).

 line number is the number of the first line of a subroutine that is to be performed when activity occurs on the specified communications channel. A line number of zero (0) disables event trapping.

Comments The ON COM(n) and COM(n) ON statements allow your program to trap communications activity. When communications activity occurs, program control is transferred to a specified subroutine. This routine usually reads in an entire message before returning back. The subroutine returns control back to the main program by executing a RETURN statement.

The ON COM(n) statement will only be executed if a COM(n) ON statement has been executed to enable event trapping. If event trapping is enabled, and if a non-zero <line number> is entered in the ON COM(n) statement, GWBasic checks between execution of each statement to see if communications activity has occurred on the specified channel. If communications activity has occurred, a GOSUB to the specified line is performed.

If a COM(n) OFF statement has been executed for the specified communications channel, the GOSUB is not performed and the event is not remembered.

If a COM(n) STOP statement has been executed for the specified communications channel, the GOSUB is not performed, but the data is remembered. The GOSUB is performed as soon as a COM(n) ON statement is executed.

When an event trap occurs, an automatic COM(n) STOP is executed so that recursive traps cannot take place. The RETURN from the trap subroutine automatically performs a COM(n) ON statement, unless a COM(n) OFF was performed inside the subroutine.

**ON COM
Statement**

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

Event trapping does not take place when GWBASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

```
Example  20 ON COM(1) GOSUB 1000
         30 COM(1) ON
         .
         .
         .
        1000 REM COMMUNICATIONS SUBROUTINE
         .
         .
         .
        1130 RETURN
```

This example shows how to set up a communications event trap. In this case, the trapping subroutine is located in lines 1000 to 1130.

**ON ERROR
Statement**

Purpose To enable error trapping and to specify the first line of the error handling subroutine.

Format ON ERROR GOTO <line number>

 line number is the first line of the error handling subroutine. If <line number> does not exist, an "Undefined line number" error message is displayed.

Comments Once error trapping has been enabled, all errors detected (including direct mode errors) will cause a jump to the specified error handling routine. The RESUME statement is used to return control back to the statement that caused the error.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors result in an error message being displayed and execution halted. An ON ERROR GOTO 0 statement that appears in an error handling routine causes GWBASIC to stop and display the error message for the error that caused the trap. It is recommended that all error handling routines execute an ON ERROR GOTO 0 to handle any errors your routine does not have a recovery action for.

Example 10 ON ERROR GOTO 1000
 :
 :
 :
 140 LPRINT
 :
 :
 :
 1000 '**** ERROR HANDLING SUBROUTINES *****
 1010 IF ERR = 27 THEN PRINT "CHECK PRINTER" : GOTO 1100
 :
 :
 :
 1090 ON ERROR GOTO 0
 1100 PRINT "PRESS ANY KEY TO CONTINUE"
 1110 A\$ = ""
 1120 WHILE A\$ = "" : A\$ = INKEY\$: WEND
 1130 RESUME

**ON ERROR
Statement**

This example shows a possible format for using an error handling routine. Line 10 enables the error trapping and specifies line 1000 as the first line of the error handling routine. If line 140 was executed and the printer had not been turned on, the error handling routine would display the message "CHECK PRINTER" and then wait for the user to press a keyboard key. After a key is pressed, control is returned to line 140. If a recovery action has not been indicated for the current error when line 1090 is executed, an error message is displayed and the program is terminated.

Notes

If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

**ON...GOSUB and ON...GOTO
Statements**

Purpose To branch to one of several specified line numbers,
 depending on the value of an expression.

Format ON x GOTO <line>[,<line>]...

 ON x GOSUB <line>[,<line>]...

x is a numeric expression in the range 0 to
 255. x is used to select which of the
 following <line> entries to branch to. (If x
 is not an integer, it is rounded to an
 integer before being used for selection.)

line is a line number in your program to branch
 to.

Comments The value of x determines which line number in the list
 will be used for branching. For example, if the value
 is three, branching occurs to the third line number in
 the list. If the value of x is zero (0) or greater than
 the number of line numbers in the list, GWBASIC
 continues with the next executable statement. If the
 value of x is outside of the specified range, and
 "illegal function call" error message is displayed.

 In the ON...GOSUB statement, each line number in the
 list must be the first line number of a subroutine
 which will eventually execute a RETURN statement to
 return control to the statement following the
 ON...GOSUB statement.

Example 150 CLS : LOCATE 3,1 '** DISPLAY MENU **
 160 PRINT TAB(35) "SELECTION MENU" : PRINT
 170 PRINT TAB(30) "1 - ENTER NEW ITEM"
 180 PRINT TAB(30) "2 - EDIT ITEM"
 190 PRINT TAB(30) "3 - DISPLAY LIST"
 200 PRINT TAB(30) "4 - END PROGRAM" : PRINT
 210 PRINT TAB(30);: INPUT "ENTER SELECTION";ANSWER\$
 220 ANSWER = VAL(ANSWER\$)
 230 ON ANSWER GOSUB 300,800,1200
 240 IF ANSWER = 4 THEN END ELSE GOTO 150

 This program segment shows a possible format for a menu
 routine. The menu is displayed and the user is asked
 for a selection. If an invalid selection is entered,
 control is returned to the beginning of the routine.
 If ANSWER equals 1, 2, or 3, branching occurs to the
 proper subroutine. Upon return from the subroutines,
 line 240 transfers control back to line 150 to
 redisplay the menu.

**ON KEY(n)
Statement**

Purpose To specify the first line number of a subroutine to be performed when a specified key is pressed.

Format ON KEY(n) GOSUB <line number>

n is a numeric expression in the range 1 to 20 indicating the key to be trapped.

- 1 - 10 function keys F1 to F10
- 11 Cursor Up
- 12 Cursor Left
- 13 Cursor Right
- 14 Cursor Down
- 15 - 20 user defined, using the KEY statement

line number is the number of the first line of a subroutine that is to be performed when the specified key is pressed. A line number of zero (0) disables event trapping.

Comments The ON KEY(n) and KEY(n) ON statements provide key trapping. Key trapping allows your program to transfer control to a subroutine if a specified keyboard key is pressed.

The ON KEY(n) statement specifies the first line number of a subroutine to transfer control to when the trapped key is pressed. The GOSUB portion of the ON KEY(n) statement will only be executed if a KEY(n) ON statement has been executed to enable key trapping for that key. If key trapping is enabled, and if the <line number> in the ON KEY(n) statement is not zero, GWBASIC checks between execution of each statement to see if the specified key has been pressed. If so, the GOSUB is performed to the specified line.

If a KEY(n) OFF statement has been executed for the specified key, the GOSUB is not performed and the key press is not remembered.

If a KEY(n) STOP statement has been executed for the specified key, the GOSUB is not performed but the key press is remembered. The GOSUB is performed as soon as a KEY(n) ON statement is executed.

**ON KEY(n)
Statement**

When a keytrap occurs and the GOSUB is performed, an automatic KEY(n) STOP is executed so that recursive traps cannot take place. The RETURN from the trap subroutine automatically perform a KEY(n) ON statement, unless an explicit KEY(n) OFF was performed inside the subroutine.

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

Event trapping does not take place when GWBasic is not executing a program, and event trapping is automatically disabled when an error trap occurs.

```
Example 10 FOR INDEX = 1 TO 9 : KEY INDEX , "" : NEXT INDEX
        20 KEY 10, "EXIT" : KEY ON
        30 ON KEY(10) GOSUB 1000 : KEY(10) ON
        40 KEY 20,CHR$(&H4)+CHR$(70) 'TRAP CTRL/BREAK
        50 ON KEY(20) GOSUB 950 : KEY(20) ON
        .
        .
        .
        950 'CTRL/BREAK KEY TRAP ROUTINE
        960 RETURN
        1000 'F10 KEY TRAP ROUTINE
        1010 CLOSE : RETURN 1020
        1020 SYSTEM
```

In this example, the FOR...NEXT loop in line 10 disable soft keys 1 to 9. Line 20 defines soft key 10 to say "EXIT" and turns on the soft key display. Line 30 defines and enables the key trap for function key F10. The subroutine at line 1000 for the F10 key trap closes all open files and then provides the exit from the program and GWBasic, returning the user to TeleDOS. Lines 40 and 50 define key 20 as the <Ctrl>/<Break> key sequence. The key trap routine for key 20 is simply a return; this prevents a user from using the <Ctrl>/<Break> sequence to end your program.

Notes The key press causing a key trap is destroyed, and therefore cannot be tested by an INPUT\$ or INKEY\$ statement.

**ON PEN
Statement**

Purpose To specify the first line number of a subroutine to be performed when the light pen is activated.

Format ON PEN GOSUB <line number>

line number is the number of the first line of a subroutine that is to be performed when the light pen is activated. A line number of zero (0) disables event trapping.

Comments A PEN ON statement must be executed to activate the ON PEN statement. After the PEN ON statement, and if a non-zero <line number> was entered in the ON PEN statement, GWBASIC checks between execution of each statement to see if the light pen has been activated. If the light pen has been activated, a GOSUB is performed to the specified line.

If a PEN OFF statement has been executed, the GOSUB is not performed and the event is not remembered.

If a PEN STOP statement has been executed, the GOSUB is not performed, but the event is remembered. The GOSUB is performed as soon as a PEN ON statement is executed.

When an event trap occurs and the GOSUB is performed, an automatic PEN STOP is executed so that recursive traps cannot take place. The RETURN from the trap subroutine automatically performs a PEN ON statement, unless an explicit PEN OFF was performed inside the subroutine.

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

Event trapping does not take place when GWBASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

**ON PEN
Statement**

```
Example    10 ON PEN GOSUB 400  
          20 PEN ON  
          .  
          .  
          .  
          400 '*** LIGHT PEN TRAP ROUTINE ***  
          .  
          .  
          .  
          490 RETURN
```

This example sets up a trap routine for the light pen.

**ON PLAY(n)
Statement**

Purpose To allow continuous background music to play during program execution.

Format ON PLAY(n) GOSUB <line number>

n is an integer expression in the range 1 to 32 indicating at what point the event trap should occur.

line number is the line number of the first line of a subroutine that is to be performed when the PLAY event trap occurs. A line number of zero (0) disables event trapping.

Comments A PLAY ON statement must be executed to activate the ON PLAY(n) statement. After the PLAY ON, and if a non-zero <line number> was entered in the ON PLAY(n) statement, GWBASIC checks between execution of each statement to see if the Music Background buffer has gone from n to n-1 notes. If so, the GOSUB portion of the ON PLAY(n) statement is performed.

When the PLAY trap occurs and the GOSUB is performed, an automatic PLAY STOP is executed so that recursive traps cannot take place. The RETURN from the trap subroutine automatically performs an PLAY ON statement, unless an explicit PLAY OFF was performed inside the subroutine.

A PLAY event trap only occurs when PLAY is in the Music Background (MB) mode.

A PLAY event trap does not occur if the Music Background buffer is empty when the PLAY ON statement is executed.

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

Event trapping does not take place when GWBASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

**ON PLAY(n)
Statement**

```
Example  10 ON PLAY(4) GOSUB 1000
         20 GOSUB 1000
         30 PLAY ON
         .
         .
         .
        1000 '*** BACKGROUND MUSIC ***
         .
         .
         .
        1090 RETURN
```

This example sets up a trap routine which is activated when less than four notes are left in the music background buffer. Line 20 initially fills the buffer, so that when line 30 is executed event trapping will take place.

**ON STRIG(n)
Statement**

Purpose To specify the first line number of a subroutine to be performed when a joystick trigger is pressed.

Format ON STRIG(n) GOSUB <line number>

n is the number of a joystick trigger as follows:

0	button A1
2	button B1
4	button A2
6	button B2

line number is the number of the first line of a subroutine that is to be performed when the joystick trigger is pressed. A <line number> of zero (0) disables event trapping.

Comments A STRIG(n) ON statement must be executed to activate the ON STRIG(n) statement. After a STRIG(n) ON statement, and if a non-zero <line number> was entered in the ON STRIG(n) statement, GWBASIC checks between execution of each statement to see if the joystick trigger has been pressed. If it has, a GOSUB is performed to the specified line.

If a STRIG(n) OFF statement has been executed, the GOSUB is not performed and is the event not remembered.

If a STRIG(n) STOP statement has been executed, the GOSUB is not performed, but the event is remembered. The GOSUB is performed as soon as a STRIG(n) ON statement is executed.

When an event trap occurs and the GOSUB is performed, an automatic STRIG(n) STOP is executed so that recursive traps cannot take place. The RETURN from the trap subroutine automatically performs a STRIG(n) ON statement, unless an explicit STRIG(n) OFF was performed inside the subroutine.

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

**ON STRIG(n)
Statement**

Event trapping does not take place when GWBasic is not executing a program, and event trapping is automatically disabled when an error trap occurs.

```

Example  10 ON STRIG(0) GOSUB 400
         20 STRIG(0) ON
           .
           .
         400 '*** STRIG(0) SUBROUTINE ***
           .
           .
         520 RETURN
    
```

This example sets up a trap routine for the button on the first joystick.

**ON TIMER
Statement**

Purpose To provide an interval timer allowing program control to be transferred to subroutine after a specified number of seconds.

Format ON TIMER(n) GOSUB <line number>

TIMER ON
 TIMER OFF
 TIMER STOP

n is a numeric expression in the range 1 to 86400 indicating the number of seconds between event traps (1 second to 24 hours). If a value for n is entered outside of the specified range, an "Illegal function call" error message is displayed.

line number is the line number of the first line of a subroutine to be executed when the event trap occurs. A <line number> of zero (0) disables event trapping.

Comments A TIMER ON statement must be executed to activate the ON TIMER(n) statement. After a TIMER ON, and if a non-zero <line number> was entered in the ON TIMER(n) statement, GWBASIC checks between execution of each statement to see if n seconds have elapsed since the last TIMER event trap. If so, the GOSUB portion of the ON TIMER(n) statement is executed.

If a TIMER OFF statement is executed, no TIMER event trapping takes place and the event is not remembered.

If a TIMER STOP statement is executed, TIMER event trapping is disabled, but the event is remembered. The GOSUB portion of the ON TIMER(n) statement is executed as soon as a TIMER ON statement is executed.

When a TIMER event trap occurs and the GOSUB is performed, an automatic TIMER STOP is executed so that recursive traps cannot take place. The RETURN from the trap subroutine automatically performs a TIMER ON statement, unless an explicit TIMER OFF was performed inside the trap subroutine.

**ON TIMER
Statement**

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

TIMER event trapping does not take place when GWBASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

```
Example 10 ON TIMER(60) GOSUB 1000
        20 TIMER ON
        30 LOCATE 1,70 : PRINT TIME$
          .
          .
          .
        1000 OLDROW = CSRLIN 'SAVE CURRENT ROW
        1010 OLDCOL = POS(0) 'SAVE CURRENT COLUMN
        1020 LOCATE 1,70 : PRINT TIME$
        1030 LOCATE OLDROW,OLDCOL
        1040 RETURN
```

This example sets up a TIMER event trap to occur every minute to update the time display in the upper-right corner of the screen.

**OPEN
Statement**

Purpose To allow input and output (I/O) to a file or device.

Format OPEN <filespec>[FOR <mode>] AS [#]<file number>
[LEN=<record length>]

or

OPEN <mode>,[#]<file number>,<filespec>
[,<record length>]

filespec is a string expression indicating the file to be opened. Refer to Chapter 4 for information on valid file specifications.

mode is one of the following:

	format 1	format 2	
--	----------	----------	--

OUTPUT	O	Specifies sequential output mode.
--------	---	-----------------------------------

INPUT	I	Specifies sequential input mode.
-------	---	----------------------------------

APPEND	A	Specifies sequential output mode and sets the file pointer at the end of file and the record number as the last record of the file. A PRINT# or WRITE# statement will then extend (append) the file.
--------	---	--

*	R	Specifies random input/output mode.
---	---	-------------------------------------

* If <mode> is omitted, the default random access mode is assumed.

file number is an integer expression in the range 1 to 15 (or the maximum number of files allowed, five by default). The file number is associated with the file for as long as it is OPEN and is used by other disk I/O statements to refer to the file.

**OPEN
Statement**

record length is an integer expression in the range 1 to 32767 that sets the record length for random files. The default record length is 128 bytes. If the /I and /S: parameters were used when GWBasic was started, <record length> cannot exceed the maximum value set with the /S: parameter.

Comments A disk file must be opened before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer.

The <file number> assigned to the file in the OPEN statement is used by the following I/O statements to refer to the file:

PRINT#	WRITE#
PRINT# USING	INPUT\$
INPUT#	GET
LINE INPUT#	PUT

A file can be opened under more than one file number at the same time, with the exceptions of 1) a sequential file can only be opened for output under one file number, 2) a sequential file cannot be opened for output if the file is currently open.

If a file is opened for input and the file does not exist, a "File not found" error message is displayed. If a file number or record number value is entered outside of the specified range, an "Illegal function call" error message is displayed.

Examples 10 OPEN "B:INVENTORY" FOR INPUT AS #1
10 OPEN "I",1,"B:INVENTORY"

These examples show the two forms of the OPEN statement that could be used to open file INVENTORY on drive B for sequential input.

40 OPEN "B:SALES\SALES.DAT" AS #1 LEN = 256

This example open the file SALES.DAT in directory SALES on drive B for random access with a record length of 256 bytes.

**OPEN COM
Statement**

Purpose To open and initialize a communications channel for input/output (I/O).

Format OPEN "COMn: [<speed>][,<parity>][,<data>][,<stop>][,RS][,CS[n]][,DS[n]][,CD[n]][,LF][,PE]]]"
AS [#]<device number> [LEN=number]

n is 1 or 2, indicating the communications channel to be opened.

speed is the baud rate, in bits per second, of the device to be opened. Valid entries are 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600. If omitted, a default value of 300 is assumed.

parity designates the parity of the device to be opened. Valid entries are:

- N None. No transmit parity, no receive parity checking.
- E Even. Even transmit parity, even transmit parity checking.
- O Odd. Odd transmit parity, odd receive parity checking.
- S Space. The parity bit is always transmitted and received as a space (0 bit).
- M Mark. The parity bit is always transmitted and received as a mark (1 bit).

If omitted, a default value of even is assumed.

data designates the number of data bits. Valid entries are: 4, 5, 6, 7, or 8. If omitted, a default value of 7 is assumed. If you enter a value of 4, the N response for <parity> is invalid. If you enter a value of 8, you must enter a <parity> response of N. If you plan to transmit or receive numeric information, you must specify a <data> value of 8 data bits because GWBasic uses all 8 bits in a byte to store numbers.

**OPEN COM
Statement**

stop indicates the number of stop bits. Valid entries are 1 or 2. The default value is 2 stop bits for baud rates of 75 or 110, 1 stop bit for all other baud rates. If a value of 4 or 5 is used for the number of data bits, a value of 2 stop bits results in 1 1/2 stop bits being used.

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, or sometimes called Received Line Signal Detect,RLSD).

n specifies the the number of milliseconds to wait for the signal before a "Device Timeout" error message is displayed. n must be in the range 0 to 65535. If n is omitted or equal to zero (0), the line status is not checked. The default values are CS1000, DS1000, and CD0.

LF specifies that a linefeed is to be sent after each carriage return.

PE enables parity checking.

device number is the number to be associated to the communications channel for future I/O statements to the device.

number is the maximum number of bytes that can be read from the communications buffer when using the GET or PUT statements. The default value is 128 bytes.

Comments The OPEN COM statement must be executed before a device can be used for RS-232 communication.

Syntax errors in the OPEN COM statement result in a "Bad File name" error message being displayed. The incorrect parameter is not shown.

A "Device timeout" error message is displayed if Data Set Ready (DSR) is not detected.

**OPEN COM
Statement**

The <speed>, <parity>, <data>, and <stop> options must be listed in the order shown in the above format. The RS, CS, DS, CD, LF, or PE options may be listed in any order, but they must be listed after the <speed>, <parity>, <data>, and <stop> options.

LF allows communication files to be printed on a serial line printer. When LF is specified, a linefeed character (0AH) is automatically sent after each carriage return character (0CH). This includes the carriage return sent as a result of the width setting. Note that INPUT# and LINE INPUT#, when used to read from a COM file that was opened with the LF option, stop when they see a carriage return, ignoring the linefeed.

If parity checking is enabled with the PE option, a "Device I/O error" error message is displayed when a parity error occurs. If there are 7 or less data bits, the high order bit is turned on. The PE option does not affect framing and overrun errors. These errors always turn on the high order bit and display the "Device I/O error" error message.

Example 10 OPEN "COM1:9600,N,8,1" AS #2

This example opens communications channel 1 at a speed of 9600 baud with no parity bit, 8 data bits, and 1 stop bit. Other program lines may now access channel 1 as device number 2.

OPTION BASE

Statement

Purpose To declare the minimum value for array subscripts.

Format OPTION BASE n

n is 0 or 1.

Comments The default option base is 0. This indicates that the lowest array subscript value is zero (0). If the statement:

 OPTION BASE 1

is executed, the lowest valid array subscript value is one (1).

The OPTION BASE statement must be executed before you define or use any arrays or a "Duplicate Definition" error message is displayed.

Example 10 OPTION BASE 1

This statement changes the option base from the default value of 0, to a value of 1.

**OUT
Statement**

Purpose To send a byte to a machine output port.

Format OUT n,m

n is an integer expression in the range -32768 to 65535 representing the port number.

m is a integer expression in the range 0 to 255 representing the data to be transmitted.

Comments The OUT statement sends the character represented by ASCII character code m to output port n.

If n is negative, OUT uses the value (65536 + n) as the output port.

Example 100 OUT 12345,255

This example send a value of hex FF to output port 12345. In 8086 assembly language, this is equivalent to:

```
MOV DX,12345  
MOV AL,255  
OUT DX,AL
```

PAINT
Statement

Purpose To fill a graphics figure with a specified color or pattern.

Format Painting

PAINT (x,y) [, <color> [, <border>]

Tiling

PAINT (x,y) , <color> , <border> [, <background>]

(x,y) are the coordinates where painting is to begin within the area to be filled. Painting should always start on a non-border point.

color is a numeric expression or a string expression. If <color> is a numeric expression, it indicates the color to use for painting and must be in the range 0 to 3 in medium resolution and 0 or 1 in high resolution. In medium resolution, 0 indicates the background color, and colors 1 to 3 indicate the colors of the current palette (see the COLOR statement for information on the current palette). In high resolution, 0 indicates black, 1 indicates white.

If <color> is a string expression, paint tiling is performed.

If <color> is omitted, painting is performed using the default color, color 3 in medium resolution and color 1 in high resolution.

border is in the range 0 to 3 and identifies the border color of the area to be filled. When the <border> color is encountered, painting of the current line and direction stops. If the <border> color is not specified, <color> is used.

background is a one-byte string expression used in paint tiling.

PAINT
Statement

Comments **Painting**

The PAINT statement can be used to paint or fill the area around point (x,y) which is bordered by color <border>. The area is painted using color <color>. If <color> is omitted, the foreground color is used (3 in medium resolution, 1 in high resolution). If <border> is omitted, <border> defaults to <color>.

If point (x,y) is presently displayed in color <border>, the PAINT statement has no effect.

Since there are only two colors in high resolution, specifying <color> different from <border> will have no effect; therefore, the <border> parameter is not needed.

The PAINT statement can be used to fill any figure, but painting areas with jagged edges or painting very complex figures increases the amount of stack space the PAINT statement requires. To avoid an "Out of Memory" error when painting complex figures, you might want to use the CLEAR statement to increase the amount of stack space available.

Paint Tiling

Paint tiling allows you to fill the area around point (x,y) with the pattern specified by the <color> string. The area to be tiled is bordered by color <border>. If <border> is omitted, the foreground color is assumed (color 3 in medium resolution, color 1 in high resolution). The <color> string is a string expression in the form:

CHR\$(&Hnn) [+ CHR\$(&Hnn)]...

where up to 64 bytes (CHR\$(&Hnn) characters) can be entered. Each byte represents an 8-bit tile mask used to display points along a horizontal line of the fill area. The method used to display points is similar to line styling used with the LINE statement. The tile mask bytes are rotated as required to align the next mask with the next vertical fill line. For example, if the entire screen is to be filled using an eight-byte <color> string and the point (0,0) is the start point, the following pattern would be used:

PAINT
Statement

		increasing x -->									
		bits of the tile mask bytes									
x , y		8	7	6	5	4	3	2	1	0	
0 , 0		x	x	x	x	x	x	x	x	x	Tile byte 0
0 , 1		x	x	x	x	x	x	x	x	x	Tile byte 1
0 , 2		x	x	x	x	x	x	x	x	x	Tile byte 2
.											
.											
0 , 6		x	x	x	x	x	x	x	x	x	Tile byte 6
0 , 7		x	x	x	x	x	x	x	x	x	Tile byte 7
0 , 8		x	x	x	x	x	x	x	x	x	Tile byte 0
0 , 9		x	x	x	x	x	x	x	x	x	Tile byte 1
.											
.											

where tile byte 0 is repeated all along the x = 0 line, and tile byte 1 is repeated all along the x = 1 line, and so on.

The following formula can be used to determine which tile mask byte in the <color> string will be used on a horizontal line (y coordinate):

$$\text{tile byte mask} = y \text{ MOD } \text{LEN}(\text{<color>})$$

In high resolution (SCREEN 2), each bit in the tile mask byte is used to determine the state of a pixel on the screen. If the bit is a 1, the point is plotted.

In medium resolution (SCREEN 1), two bits are required to describe the color of a pixel; therefore, the bits of a tile mask byte determine the color of four pixels on the screen. The following table relates the mask bit patterns to pixel colors:

Screen Color	Color in Binary	Pattern to draw a solid line Binary	Hexadecimal
0	00	00000000	&H00
1	01	01010101	&H55
2	10	10101010	&HAA
3	11	11111111	&HFF

PAINT Statement

Occasionally, you may want to paint over an area that has already been painted in the color of two consecutive lines (tile mask bytes) in the tile pattern. Normally, this condition terminates the tiling, because the point being painted is surrounded by the same color. For example, you would not be able to tile alternating red and green lines over a red background. After the first red line is drawn, the tiling would be terminating.

The <background> parameter allows you to skip over a terminating condition. In the above example, entering a <background> value of CHR\$(&HAA) (a red line) would allow you to draw alternating red and green lines over the red background. PAINT returns an "Illegal function call" error message if you specify more than two consecutive lines (tile mask bytes) in the <color> string that are the same as the byte in the <background> string.

Examples

```
10 SCREEN 1,0 : COLOR 0,1 : CLS
20 CIRCLE (160,100),50,3
30 PAINT (160,100),1,3
```

This example draws a circle in the middle of the screen and uses the PAINT statement to paint it cyan.

```
10 SCREEN 2 : CLS 'HIGH RESOLUTION GRAPHICS
20 LINE (280,80)-(360,120),1,B
30 PAINT(320,100),CHR$(&H81)+CHR$(&H42)+CHR$(&H24)
  +CHR$(&H18)+CHR$(&H18)+CHR$(&H24)+CHR$(&H42)
  +CHR$(&H81)
```

This example uses the PAINT statement to fill the box in the middle of the screen with an "X" pattern.

```
10 SCREEN 1,0 : COLOR 0,1 : CLS
20 LOCATE 4,1 : PRINT "WITHOUT BACKGROUND"
30 LINE (160,10)-(220,50),3,B
40 PAINT (190,30),2,3
50 PAINT (190,30),CHR$(&H55)+CHR$(&HAA),3
60 ' *** WITH BACKGROUND TILE
70 LOCATE 17,3 : PRINT "WITH BACKGROUND"
80 LINE (160,110)-(220,150),3,B
90 PAINT (190,130),2,3
100 PAINT (190,130),CHR$(&H55)+CHR$(&HAA),3,CHR$(&HAA)
```

This example shows how the <background> parameter is used to tile a magenta background with alternating cyan and magenta lines.

**PEEK
Function**

Purpose To return a byte of data from a specified memory location.

Format PEEK(n)

n is a numeric expression in the range -32768 to 65535 representing the offset into the segment defined by the last DEF SEG statement.

Comments The PEEK function returns an integer value in the range 0 to 255 representing the ASCII code of the byte of data located at the specified memory location.

PEEK rounds n to an integer before determining the specified memory location. If n is negative, the offset (65356 + n) is used.

Example 10 screen 0,1 : width 80
 20 DEF SEG = &HB800
 30 CLS
 40 PRINT "HELLO"
 50 FOR I = 0 TO 8 STEP 2
 60 PRINT CHR\$(PEEK(I));
 70 NEXT I
 RUN
 HELLO
 HELLO
 Ok

In this example, line 20 sets the current segment to the beginning of video memory. Lines 30 and 40 clear the screen and place the word HELLO in the first 5 character positions on the screen. The FOR...NEXT loop in lines 50 to 70 use the PEEK statement to print out the characters in the first five character positions of video memory.

Notes PEEK is the complementary function of the POKE statement.

**PEN
Function**

Purpose To read the light pen.

Format PEN(n)

- n is a numeric expression in the range 0 to 9 determining the value returned by the PEN function as follows:
- 0 A flag indicating whether the pen was down since the last poll. Returns -1 if down, 0 if not.
 - 1 Returns the x coordinate of the pixel where the light pen was last pressed (0 - 319 in medium resolution, 0 - 639 in high resolution).
 - 2 Returns the y coordinate of the pixel where the light pen was last pressed (0 - 199 in both medium and high resolution).
 - 3 Returns -1 if the light pen is down, 0 if it is up.
 - 4 Returns the last known valid x pixel coordinate (0 - 319 in medium resolution, 0 - 639 in high resolution).
 - 5 Returns the last known valid y pixel coordinate (0 - 199 in both medium and high resolution).
 - 6 Returns the character row position where the light pen was last pressed (1 to 24).
 - 7 Returns the character column position where the light pen was last pressed (1 - 40 in 40-column mode, 1 - 80 in 80-column mode).
 - 8 Returns the last known character row where the light pen was positioned (1 - 24).
 - 9 Returns the last known valid character column where the light pen was positioned (1 - 40 in 40-column mode, 1 - 80 in 80-column mode).

**PEN
Function**

Comments A PEN ON statement must be executed before a pen read can be made using the PEN function.

When the light pen is in the border area of the screen, the coordinate values returned by the PEN function are not accurate.

Example 10 CLS
 20 PEN ON
 30 P = PEN(3)
 40 LOCATE 1,1 : PRINT "PEN IS";
 50 IF P THEN PRINT "DOWN" ELSE PRINT "UP"
 60 GOTO 20

This example produces an endless loop to print the current pen switch status (UP/DOWN).

**PEN
Statements**

Purpose To enable the light pen read function and event trapping .

Format PEN ON

 PEN OFF

 PEN STOP

Comments The light pen is initially off. A PEN ON statement must be executed before a pen read using the PEN function. The PEN ON statement also enables event trapping using the ON PEN statement.

PEN OFF disables the light pen read function and event trapping. If action by the light pen occurs, the event is not be remembered.

PEN STOP disables event trapping of the light pen, but if an event occurs it is remembered. The ON PEN statement is executed as soon as a PEN ON statement is executed.

Example 10 PEN ON

This statement enables event trapping and allows the PEN function to be used.

PLAY

Function

Purpose To return the number of notes currently in the music background buffer.

Format PLAY(n)

n is a dummy argument that can be any value.

Comments Play(n) only returns the number of notes in the music buffer when in the Music Background (MB) mode. The value returned ranges from 0 to 32, which is the maximum number of notes the buffer can hold.

PLAY(n) returns zero (0) when your program is running in the Music Foreground (MF) mode.

Example

```

150 GOSUB 1000
160 A$ = ""
170 WHILE A$ = ""
180 A$ = INKEY$
190 IF PLAY(0) = 1 THEN GOSUB 1000
200 WEND

```

```

      .
      .
      .
1000 PLAY "MB ML O2 CDEFGAB>C<BAGFEDC"
1010 RETURN

```

In this example, the PLAY function in line 190 is used to refill the music background buffer so the scale (using the PLAY statement in line 1000) plays in the background while the program waits for a key to be pressed.

PLAY
Statement

Purpose To play music.

Format PLAY <string>

string is a string expression made up of the following music commands:

A-G[#{#|+|-}] Plays a note in the range A-G. # or + after the note specifies sharp; - specifies flat.

L n Sets the length of each note. The note length is equal to 1/n. n must be an integer in the range 1 to 64 or an "Illegal function call" error message is displayed.

- L1 = whole note
- L2 = half note
- L3 = 1/3 of a 4 beat measure
- L4 = quarter note
- .
- .
- .
- L8 = eighth note
- .
- .
- L64 = sixty-fourth note

The length may follow the note if you are only changing the note length. In this case, A16 is equivalent to L16A.

Pn Sets the pause (rest) length. The length of the rest is calculated in the same way as the note length, where n can range from 1 to 64.

Dotted note. When a period (.) is placed after a note or rest, the length is multiplied by 3/2. If multiple periods are placed after a note or pause, the length is scaled accordingly. For example, A. is 3/2, A.. is 9/4, and A... is 27/4 as long as A.

PLAY
statement

- On** Octave. Sets the current octave. There are seven octaves numbered 0 through 6. Each octave ranges from C to B, with octave 3 starting at middle C. Octave 4 is the default octave.
- >** Increment octave. The greater than symbol (>) increments the current octave by one. Increment octave will not increment above octave 6.
- >>** Increment octave twice. Two greater than symbols together (>>) increments the current octave twice. Increment twice will not increment above octave 6.
- <** Decrement octave. The less than symbol (<) decrements the current octave by one. Decrement octave will not decrement below octave 0.
- <<** Decrement octave twice. Two less than symbols together (<<) decrements the current octave twice. Decrement twice will not decrement below octave 0.
- Nn** Play note n, where n is in the range 0 to 84. In the seven octaves, there are 84 possible notes; N1 is the same as C in octave 0, and N84 is the same as B in octave 6. N0 is a rest. The N subcommand provides an alternate way of selecting notes.
- Tn** Sets the tempo of the music. n indicates the number of quarter beats per minute and may range from 32 to 255. The default value is 120 beats per minute.
- MF** Music foreground. Sets music created by the PLAY and SOUND statements to run in the foreground. That is, each subsequent note or sound will not start until the previous note or sound has finished. This is the default setting.

PLAY
Statement

- MB** Music background. Sets music created by the **PLAY** and **SOUND** statements to run in the background. That is, each note or sound is placed in a buffer allowing the your program to continue executing while the note or sound plays in the background. Up to 32 notes or rests can be played in the background at one time.
- MN** Music normal. Each note is played $7/8$ of the note length specified by the **Ln** subcommand. This is the default setting.
- ML** Music legato. Each note is played the full note length specified by the **Ln** subcommand.
- MS** Music staccato. Each note is played $3/4$ of the note length specified by the **Ln** subcommand.
- X<variable>;** Play a substring. The **X** subcommand allows you to assign a string of music subcommands to a string variable and then use the **X** subcommand with the variable name in the **PLAY** command. A semicolon (;) must follow the variable name.

Comments **PLAY** uses a concept similar to that used in the **DRAW** statement, by embedding a music macro language into one statement. A set of subcommands, used as part of the **PLAY** statement itself, specifies how music is played.

The **n** argument used in the subcommands can be a constant like **6**, or it can be a variable used in the **=variable;** format, where the equal sign (=) and the semicolon (;) are required.

Spaces embedded within the strings are ignored. A semicolon may be used to separate subcommands within a string, except after the **MF**, **MB**, **MN**, **ML**, and **MS** subcommands.

**PLAY
Statement**

```

Example 10 'MICHELLE BY J. LENNON & P. MCCARTNEY
        20 REPEAT$ = "L2AA P4L4B-L2F L4EAAE DFA-F L2E"
        30 OPENING$ = "XREPEAT$; L4DF L1E"
        40 CENTER1$ = "XREPEAT$; L6DEF L2E.L4A L6>DC<A>DC<A
          L4>EL2D. P8<L8AB-AL4B-F L1F"
        50 CENTER2$ = "P8L8AAAL4>D<A ML GFL8FMNFL4G AAAA AAAA
          A2L4GF L1E"
        60 CENTER$ = CENTER1$ + CENTER2$
        70 ENDING$ = "XREPEAT$; L6DEF L2E.L8DE L4FDGE GDG.L8E
          L2FL4ED L2C#L4DE L1D"
        80 PLAY "MN T100 O3 XOPENING$; XCENTER$; XENDING$;"
    
```

This example plays the song Michelle by John Lennon and Paul McCartney. Spaces were used between measures within the subcommand strings to make it easier to read and check against the actual music. Line 20 assigns the string variable REPEAT\$ to a string of subcommands representing a portion of the song that is repeated in three places. This saves program space and reduces the chance of making a typing mistake. Line 30 uses the X subcommand to incorporate REPEAT\$ into the opening section of the song. Line 40 uses the < and > symbols to move between notes in the third and fourth octave. Line 70 is the line that actually plays the song. It first sets the music to music normal, then sets the tempo at 100 quarter notes per minute, sets the current octave to octave 3, and finally plays the three sections of the song, OPENING\$, CENTER\$, and ENDING\$.

**PLAY ON, PLAY OFF, and PLAY STOP
Statements**

Purpose To enable event trapping for the ON PLAY statement.

Format PLAY ON

 PLAY OFF

 PLAY STOP

Comments These statements are used to enable and disable the event trapping of the ON PLAY statement. The ON PLAY statement is used to provide continuous background music during program execution.

The PLAY ON statement enables event trapping by the PLAY ON statement. If a non-zero line number was entered in the ON PLAY statement, GWBASIC checks between execution of each statement to see if the music buffer has gone from n to n-1 notes. If so, the GOSUB portion of the ON PLAY statement is performed.

The PLAY OFF statement disables the event trapping. If the play event takes place, the event is not remembered.

The PLAY STOP statement disables the event trapping, but a play event is remembered. The GOSUB portion of the ON PLAY statement is executed when PLAY ON is executed.

Example 10 ON PLAY(4) GOSUB 1000
 20 GOSUB 1000
 30 PLAY ON
 .
 .
 .
 1000 'BACKGROUND MUSIC
 .
 .
 .
 1090 RETURN

This example sets up a trap routine which is activated when less than four notes are left in the music background buffer.

**PMAP
Function**

Purpose To map the specified coordinate to either its screen coordinate or its coordinate in the Cartesian coordinate system in real space.

Format PMAP (x,n)

x is the coordinate to be mapped.

n is an integer expression in the range 0 to 3 indicating the mapping to be performed.

0 maps the x coordinate of a point in the Cartesian coordinate system to its physical position on the screen.

1 maps the y coordinate of a point in the Cartesian coordinate system to its physical position on the screen.

2 maps the x coordinate of a point on the screen to its position in the Cartesian coordinate system.

3 maps the y coordinate of a point on the screen to its position in the Cartesian coordinate system.

Comments The PMAP function is used to translate coordinates between their position in the Cartesian coordinate system as defined by the WINDOW statement and their position on the screen as described by the VIEW statement.

Example 10 SCREEN 2 : CLS
 20 WINDOW (-100,-100)-(100,100)
 30 PRINT "THE CENTER OF THE CARTESIAN COORDINATE"
 40 PRINT "SYSTEM FOR WINDOW (-100,-100)-(100,100)"
 50 PRINT "IS LOCATED AT SCREEN COORDINATES (";
 60 PRINT PMAP(0,0) " , " PMAP(0,1) ")"
 RUN
 THE CENTER OF THE CARTESIAN COORDINATE
 SYSTEM FOR WINDOW (-100,-100)-(100,100)
 IS LOCATED AT SCREEN COORDINATES (320 , 99)
 Ok

This example uses the PMAP function to find the screen coordinates of point (0,0) in the Cartesian coordinate system for a WINDOW of (-100,-100)-(100,100).

**POINT
Function**

Purpose To return the color value of a pixel on the screen or the current graphics coordinate. For use in the graphics mode only.

Format POINT(x,y)
POINT(n)

(x,y) are the coordinates of a pixel on the screen to return the color value for. Coordinates are specified in absolute form.

n is a number from 0 to 3 returning values as indicated below:

- 0 returns the current physical screen x coordinate.
- 1 returns the current physical screen y coordinate.
- 2 returns the current x coordinate in the Cartesian coordinate system if WINDOW is active. If WINDOW is not active, 2 returns the current physical x coordinate.
- 3 returns the current y coordinate in the Cartesian coordinate system if WINDOW is active. If WINDOW is not active, 3 returns the current physical y coordinate.

Comments In the first format, POINT returns the color of the specified pixel. If the specified coordinates are out of range, POINT returns a value of -1. In medium resolution, POINT returns a value of 0 to 3, where 0 indicates the background color and 1 to 3 are the three colors from the current palette. In high resolution, POINT returns a value of 0 or 1. 0 indicates black, 1 indicates white.

In the second format, POINT returns the specified coordinate. Physical coordinates refer to actual screen coordinates as described in Chapter 4. Coordinates in the Cartesian coordinate system refer to coordinates in real space (refer to the WINDOW statement).

POINT
Statement

```
Examples 100 FOR I = 0 TO 319
          110 IF POINT(I,100)=0 THEN PSET(I,100) ELSE
            PRESET(I,100)
          120 NEXT I
```

This example draws a horizontal line across the screen at y equal to 100. If the point to be drawn is presently in the background color, the point is drawn in color 3. If the point to be drawn is presently in one of the three colors from the current palette, the point is drawn in the background color.

```
10 SCREEN 1,0 : CLS
20 PRINT "SCREEN CONDITION";
30 PRINT TAB(26) "X" TAB(36) "Y" : PRINT
40 PRINT "NORMAL"; : GOSUB 100
50 WINDOW (0,0)-(319,199)
60 PRINT "WINDOW ACTIVE"; : GOSUB 100
70 WINDOW SCREEN (0,0)-(319,199)
80 PRINT "WINDOW SCREEN ACTIVE"; : GOSUB 100
90 END
100 PRESET (0,0)
110 PRINT TAB(25) POINT(0) TAB(35) POINT(1)
120 PRINT : RETURN
```

```
RUN
SCREEN CONDITION           X           Y

NORMAL                     0           0

WINDOW ACTIVE              0          199

WINDOW SCREEN ACTIVE      0           0
```

Ok

This example illustrates the values returned by the POINT function. Notice the difference in values returned when the WINDOW is active with and without the SCREEN argument.

**POKE
Statement**

Purpose To write a byte of data into a memory location.

Format POKE n,m

n is a numeric expression in the range -32768 to 65535 representing an address offset into the current segment defined by the last DEF SEG statement.

m is a numeric expression in the range 0 to 255 representing the ASCII code for the data to be written into memory.

Comments The POKE statement can be used to load machine language subroutines into memory, or passing arguments and results to and from machine language subroutines.

POKE rounds n and m to integers before the write operation is performed. If n is negative, the offset (65536+n) is used.

Example 10 CLS 'CLEAR SCREEN
 20 'SET THE SEGMENT TO VIDEO MEMORY AT HEX B800
 30 DEF SEG = &HB800
 40 'PRINT SMILEY FACES ON LINE 1
 50 FOR INDEX = 0 TO 160 STEP 4
 60 POKE INDEX,1
 70 NEXT INDEX
 80 'RESET SEGMENT TO THE GWBASIC DATA SEGMENT
 90 DEF SEG
 100 PRINT

This example uses the POKE statement to place ASCII character 1 (smiley face) into every other character position on the top line of the display memory.

Notes GWBASIC does not do any checking on the addresses you are POKEing into. Make sure you do not POKE into GWBASIC's workarea, or you could cause your program or GWBASIC to crash.

The POKE statement is the complementary function to the PEEK function.

**POS
Function**

Purpose To return the current horizontal (column) position of
the cursor.

Format POS(0)

n is a dummy argument.

Comments The POS function returns the current column position of
the cursor. In the 40-column mode, POS returns a value
in the range 1 to 40. In the 80-column mode, POS
returns a value in the range 1 to 80.

Example 50 ROW = CSRLIN 'STORE CURRENT LINE
60 COL = POS(0) 'STORE CURRENT COLUMN
70 INPUT "ENTER 5 DIGIT PART NUMBER";PARTNO\$
80 IF LEN(PARTNO\$) = 5 THEN 120
90 LOCATE ROW,COL : PRINT SPC(35)
100 LOCATE ROW,COL
110 GOTO 70

In this program segment, the POS and CSRLIN functions
are used to store the location of the first character
position of the INPUT prompt. If a 5-character
response is not entered, line number 90 erases the
prompt and the response. Line number 100, 110 ,and 70
repeat the prompt in the same place on the screen as it
originally appeared.

**PRESET
Statement**

Purpose To draw a point on the screen at the specified coordinates in the specified color.

Format PRESET(x,y),[<color>]

 (x,y) is the coordinates of the point to be drawn. Coordinates can be specified in either absolute or relative coordinates (refer to Chapter 4 for information on specifying screen coordinates).

 color is the color to be used to draw the point. In medium resolution, <color> is in the range 0 to 3, where 0 indicates the background color and 1 to 3 are the three colors of the current palette. In high resolution, <color> is 0 or 1. 0 indicates black, 1 indicates white. If <color> is omitted, the default is the background color, color 0.

Comments The PRESET statement allows you to draw points anywhere on the screen.

 If an out of range coordinate is specified, no action is taken and no error message is displayed.

Example 10 SCREEN 1 'SET MEDIUM RESOLUTION GRAPHICS
 20 COLOR 0,1 'SET BLACK BACKGROUND, PALETTE 1
 30 CLS 'CLEAR SCREEN
 40 FOR I = 3 TO 0 STEP -1
 50 FOR X = 0 TO 319
 60 Y = 100 - 80 * SIN(X * 3.141593/180 * 360/320 * 2)
 70 PRESET(X,Y),I
 80 NEXT X
 90 NEXT I

 In this example, the PRESET statement is used to draw two cycles of a sine wave. The outer FOR...NEXT loop is used to draw the sine wave in the three colors of palette 1 and then erase it by drawing it in the background color, color 0.

Notes The PRESET statement is the same as the PSET statement, except the background color is the default color.

**PRINT
Statement**

Purpose To display data on the screen.

Format PRINT [<expression>[{:|,}]<expression>...]

? [<expression>[{:|,}]<expression>]

expression is a numeric or string expression. String constants must be enclosed in quotation marks. <expression>s may be separated by spaces, commas, or semicolons. If no <expression> items are listed, a blank line is displayed.

Comments The PRINT statement is used to display data on the screen. The format of the printed data depends on the punctuation used between the list of <expression>s. GWBASIC divides the screen line into print zones of 14 columns each. If the list of <expression>s are separated by commas, the next item is printed at the beginning of the next print zone. If the list of <expression>s are separated by semicolons or spaces, the next item is printed immediately after the last item.

If a comma, semicolon, SPC function, or TAB function terminates the list of print <expression>s, the next PRINT statement begins printing on the same line, spaced accordingly. If the list of <expression>s does not terminate in a comma, semicolon, SPC function, or TAB function, a carriage return is printed at the end of the line, forcing the cursor down to the beginning of the next line.

If the length of an <expression> to be printed from the list of <expression>s (other than the first item in the list) is greater than the number of character positions remaining on the current line, the entire <expression> is printed at the beginning of the next line. If the length of an <expression> is longer than the current line width, GWBASIC prints the remaining characters of the <expression> on the next physical line.

**PRINT
Statement**

Numeric values are always printed with a trailing space. Positive numbers are preceded by a space, negative numbers are preceded by a minus sign. Single-precision numbers are printed in fixed-point format with seven or fewer digits, except where seven digits limits the accuracy, then floating-point format is used. For example, $X = 0.00000012$; X would be printed as $1.2E-07$, because eight digits are needed to represent the two digits of accuracy. Double-precision numbers are printed in fixed-point form with 16 or fewer digits, except where 16 digits limits the accuracy, then floating-point format is used.

A question mark (?) may be used in place of the word PRINT in a PRINT statement. GWBASIC replaces the question mark with the word PRINT when you list your program.

Examples

```
10 X = 5
20 PRINT 5,X+5,X-5
30 PRINT 5;X*5 X^5
RUN
5          10          0
5 25 3125
```

This example shows the different print formats resulting from separating the expressions by commas, semicolons, and spaces.

```
10 INPUT "ENTER A NUMBER";X
20 PRINT
30 PRINT X "SQUARED IS" X^2 "AND";
40 PRINT X "CUBED IS" X^3
RUN
ENTER A NUMBER? 5

5 SQUARED IS 25 AND 5 CUBED IS 125
Ok
```

This example uses the PRINT statement in line 20 to place a blank line between the INPUT prompt and the output of lines 30 and 40. The semicolon at the end of line 30 causes the PRINT statement in line 40 to print on the same line as the PRINT statement in line 30.

Notes To print data on the printer, refer to the LPRINT and LPRINT USING statements.

PRINT USING

Statement

 Purpose To print strings or numbers using a specified format.

Format PRINT USING <string expression>;<expression>
 [{;|,}<expression>...]

string expression is a string expression composed of special formatting characters. These characters determine the format of the printed strings or numbers.

expression is a string or numeric expression to be printed. The <expression>s can be separated by commas or semicolons. A "Syntax error" error message is displayed if spaces are used to separate the <expression>s.

Comments The following two sections describe the special formatting characters used in <string expression>.

String Fields

When PRINT USING is used to print string expressions, one of three formatting characters can be used to format the string field:

! The exclamation point is used to indicate that only the first character in the listed <expression>s is to be printed.

\n spaces\ The backslashes with n spaces indicate that 2+n characters from the string are to be printed. If the backslashes are printed with no spaces, only the first two characters are printed. If the field is longer than the string, the string is left-justified in the field and padded with spaces to the right.

PRINT USING Statement

& The ampersand indicates that a variable field length will be used. The string will be printed without modification.

```
10 A$ = "LOOK" : B$ = "OUT"
20 PRINT USING "!" ; A$ ; B$
30 PRINT USING "\ \ " ; A$ , B$ , "!!"
40 PRINT USING "&" ; A$ ; B$
RUN
LO
LOOKOUT !!
LOOKOUT
Ok
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

- # A number sign is used to represent a digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified in the field, the number is right-justified and preceded by spaces.
- . A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point and the number is less than 1, a 0 is printed. If the number has more digits of accuracy to the right of the decimal point than positions specified in the format field, the number is rounded to the specified number of digits.

```
PRINT USING "##.## " ; .78 ; 87.654 ; 10.2
0.78 87.65 10.20
```

Two spaces were included at the end of the string expression to separate the printed values on the line.

**PRINT USING
Statement**

- + A plus sign at the beginning or end of the format field causes the sign of the number to be printed before or after the number.

```
PRINT USING "+##.##  " ; -68.95, 22.449
-68.95  +22.45
```

```
PRINT USING "##.##+  " ; -68.95, 22.449
68.95-  22.45+
```

- A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.

```
PRINT USING "##.##-  " ; -68.95, 22.449
68.95-  22.45
```

- ** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two characters.

```
PRINT USING "***.#  " ; 12.39, -0.9, 765.1
*12.4  *-0.9  765.1
```

- \$\$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The \$\$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with \$\$.
- Negative numbers are printed with the minus sign immediately to the left of the dollar sign.

```
PRINT USING "$$##.##  " ; 456.78, 34.75, 4.25
$456.78  $34.75  $4.25
```

- **\$ The **\$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **\$ specifies three more digit positions, one of which is the dollar sign.

The exponential format cannot be used with **\$. Negative numbers are printed with the minus sign immediately to the left of the dollar sign.

```
PRINT USING "***$##.##  " ; 2.34, -26.75
***$2.34  *-$26.75
```

**PRINT USING
Statement**

, A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the beginning or end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with exponential (^^^^) format.

```
PRINT USING "####, .##";1234.5
1,234.50
```

```
PRINT USING "#,###.##";1234.5
1,234.50
```

```
PRINT USING ",####.##";1234.5
,1234.50
```

^^^^ Four carets (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
2.35E+02
```

```
PRINT USING ".####^^^^-";888888
.8889E+06
```

```
PRINT USING "+.##^^^^";123
+.12E+03
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

**PRINT USING
Statement**

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22
```

```
PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error message is displayed.

```
Example 10 AMOUNT = 10.45
        20 PRINT USING "The price is $$$##";AMOUNT
        RUN
        The price is $10.45
        Ok
```

This example shows how string constants can be added into the format string.

**PRINT# and PRINT# USING
Statements**

Purpose To write data to a sequential file.

Format PRINT #<file number>,[USING <string expression>]
<expression>[{:|,}<expression>...]

file number is the number under which the file was opened for output in the OPEN statement.

string expression is a string expression comprised of special formatting characters. These formatting characters determine the field and format of the output <expression>s. Refer to the PRINT USING statement for a description of the special formatting characters.

expression is the numeric or string expression that will be written to the file. <expression>s can be separated by commas or semicolons.

Comments PRINT# does not compress data. An image of the data is written to the file, just as it would be displayed on the terminal screen with a PRINT statement. For this reason, care should be taken to delimit (separate) the data, so that it will be input correctly.

If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the file. For example, let A = 45.6 and B = 123.78. The statement:

PRINT #1,A,B

would write the following to file:

45.6 123.78

The statement:

PRINT #1,A;B

would write the following to file:

45.6 123.78

In the second case, only the leading and trailing spaces normally printed with a number are placed in the file.

PRINT# and PRINT USING Statements

To format the string expressions correctly in the file, use explicit delimiters in the list of expressions.

For example, let A\$ = "CAMERA" and B\$ = "93604-1". The statement:

```
PRINT #1,A$;B$
```

would write CAMERA93604-1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT #1,A$;" ";B$
```

The image written to the file is

```
CAMERA,93604-1
```

which can be input as two string variables.

If the strings themselves contain commas, semicolons, significant leading spaces, carriage returns, or linefeeds, write them to the file surrounded by explicit quotation marks, CHR\$(34).

For example, let A\$ = "CAMERA, AUTOMATIC" and B\$ = " 93604-1". The statement:

```
PRINT#1,A$;B$
```

would write the following image to file:

```
CAMERA, AUTOMATIC    93604-1
```

And the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A\$ and "AUTOMATIC 93604-1" to B\$. To separate these strings properly in the file, write double quotation marks to the file image using CHR\$(34). The statement:

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to the file:

```
"CAMERA, AUTOMATIC"    93604-1"
```

**PRINT# and PRINT# USING
Statement**

The INPUT statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A\$ and " 93604-1"
to B\$.

The PRINT# statement may also be used with the USING
option to control the format of the file. For example:

```
PRINT#1,USING"$###.##,";J;K;L
```

Notes The WRITE# statement provides an easier way to write
data into a sequential file.

**PSET
Statement**

Purpose To draw a point on the screen at the specified position. Graphics mode only.

Format PSET (x,y) [,<color>]

(x,y) is the coordinates of the point to be drawn. Coordinates can be specified in either absolute or relative coordinates (refer to Chapter 4 for information on specifying screen coordinates).

color is the color to be used to draw the point. In medium resolution, <color> is in the range 0 to 3, where 0 indicates the background color and 1 to 3 are the three colors of the current palette. In high resolution, <color> is 0 or 1. 0 indicates black, 1 indicates white. If <color> is omitted, the default is the foreground color, color 3 in medium resolution, color 1 in high resolution.

Comments The PSET statement allows you to draw points anywhere on the screen.

If an out of range coordinate is specified, no action is taken and no error message is displayed.

Example 10 SCREEN 1 'SET MEDIUM RESOLUTION GRAPHICS
 20 COLOR 0,1 'SET BLACK BACKGROUND, PALETTE 1
 30 CLS 'CLEAR SCREEN
 40 FOR I = 3 TO 0 STEP -1
 50 FOR X = 0 TO 319
 60 Y = 100 - 80 * SIN(X * 3.141593/180 * 360/320 * 2)
 70 PSET(X,Y),I
 80 NEXT X
 90 NEXT I

In this example, the PSET statement is used to draw two cycles of a sine wave. The outer FOR...NEXT loop is used to draw the sine wave in the three colors of palette 1 and then erase it by drawing it in the background color, color 0.

Notes The PSET statement is the same as the PRESET statement, except the foreground color is the default color.

**PUT
Statement (Files)**

Purpose To write a record from a random buffer to a random access file.

Format PUT [#]<file number>[,<record number>]

file number is the number under which the file was opened in the OPEN statement.

record number is the number of the record into which the data is being written. If <record number> is omitted, the next available record number (after the last PUT) is used. <record number> must be in the range 1 to 32767.

Comments The LSET, RSET, PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement. If the WRITE# statement is used, GWBASIC pads the buffer with spaces up to the carriage return.

The PUT statement also allows fixed-length output to a GWBASIC COM file. In this case, the <record number> indicates the number of bytes of data to write to the communications file. This value must be less than or equal to the value set with LEN option in the OPEN COM statement.

Example 10 OPEN "B:SALES.DAT" AS #1 LEN = 4
 20 FIELD #1,4 AS AMOUNT\$
 30 INPUT "ENTER WEEK (1 - 52)";WEEK
 40 INPUT "ENTER DOLLAR SALES";AMOUNT
 50 LSET AMOUNT\$ = MKS\$(AMOUNT)
 60 PUT #1,WEEK
 70 CLOSE

In this example, the file SALES.DAT on drive B is used to hold dollar sales figures for one year. The week number is used as the record number. The INPUT statements in lines 30 and 40 input the week number (record number) and the dollar value. Line 50 converts the dollar value to a string and places it in the random file buffer. The PUT statement in line 60 writes the data to the diskette file.

Notes GWBASIC and TeleDOS block as many records as possible into a 512 byte sector; therefore, the PUT statement does not always perform a physical write to the diskette file.

**PUT
Statement (Graphics)**

Purpose To write an image array (read by the GET statement) onto a specified area of the screen.

Format PUT (x,y), <array name>[, <action verb>

(x,y) is the coordinates of the top left corner of the image being written.

array name is the name of a numeric array under which the image was saved using the GET statement.

action verb is a word describing the logic to be used to place the image on the screen. The allowable <action verb>s are:

- PSET
- PRESET
- XOR
- OR
- AND

Comments The PUT statement takes the data stored in the array by the GET statement and places it on the screen at the specified location. The PUT statement allows you to dictate how the image will interact with the present screen data through the use of an action verb. The action verbs perform as follows:

PSET Places the data on the screen exactly as it was read by the GET statement.

PRESET Places the data on the screen in a negative image of the data read by the GET statement.

		Resolution			
		Medium		High	
GET	PUT			GET	PUT
0	3			0	1
1	2			1	0
2	1				
3	0				

**PUT
Statement (Graphics)**

XOR Places the image on the screen using a logic similar to Boolean XOR logic. XOR is useful for animation, because an image XOR'd on the screen can be XOR'd back off and the background is restored to its original state. The XOR action verb is the default.

		array value				
		0	1	2	3	
s	---	+	-----	-----	-----	
c	0		0	1	2	3
r	1		1	0	3	2
e	2		2	3	0	1
e	3		3	2	1	0
n						

OR Superimposes the image onto the existing screen image using a logic similar to Boolean OR logic.

		array value				
		0	1	2	3	
s	---	+	-----	-----	-----	
c	0		0	1	2	3
r	1		1	1	3	3
e	2		2	3	2	3
e	3		3	3	3	3
n						

AND Places the image on the screen using a logic similar to the Boolean AND logic.

		array value				
		0	1	2	3	
s	---	+	-----	-----	-----	
c	0		0	0	0	0
r	1		0	1	0	1
e	2		0	0	2	2
e	3		0	1	2	3
n						

the PUT statement with the XOR action verb.

1. Draw the image on the screen.
2. Read the image into an array using the GET statement.

PUT

Statement (Graphics)

3. Calculate the next screen position for the image.
4. Erase the image from the present screen position using the PUT statement (with XOR).
5. Place the image in the new screen position using the PUT statement (with XOR).
6. Loop back to step 3.

Movement done this way will leave the background unchanged. Flicker can be reduced by minimizing the time between steps 4 and 5, and by making sure there is enough time delay between steps 5 and 4. If more than one object is being animated, every object should be processed at once, one step at a time.

Example (Refer to the example listed in the GET statement.)

**RANDOMIZE
Statement**

Purpose To reseed the random number generator.

Format RANDOMIZE [n]

n is any numeric expression. n is used as the random number seed.

Comments If n is omitted, GWBASIC suspends program execution and requests a value by displaying:

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded using the RANDOMIZE statement, the RND function returns the same sequence of random numbers each time your program is run. Placing a RANDOMIZE statement at the beginning of the program and changing the argument with each run will produce a new sequence of random numbers each time.

The TIMER function can be used as the argument of the RANDOMIZE statement to produce a new random number sequence from within your program.

Example 10 RANDOMIZE
 20 FOR I=1 TO 5
 30 PRINT INT(10*RND+1);
 40 NEXT I
 RUN
 Random Number Seed (-32768 to 32767)? 3
 3 6 3 3 1
 Ok
 RUN
 Random Number Seed (-32768 to 32767)? 5
 1 10 9 4 6
 Ok
 RUN
 Random Number Seed (-32768 to 32767)? 3
 3 6 3 3 1
 Ok

This example produces random numbers between 1 and 10. Note that using the same number again produced the same random number sequence.

10 RANDOMIZE TIMER

This line uses the TIMER function with the RANDOMIZE statement to seed the random number generator.

**READ
Statement**

Purpose To read values from a DATA statement and assign them to variables.

Format READ <variable>[,<variable>]...

variable is a string or numeric variable or an array element which will receive the next value in the DATA table.

Comments A READ statement must always be used in conjunction with a DATA statement. READ statements assign values from the DATA statements to the variables in the READ statement on a one-to-one basis. The variable type (string or numeric) used in the READ statement must agree with the next value in the DATA table or a "Syntax error" error message is displayed.

A single READ statement may access one or more DATA statements, or several READ statements may access the same DATA statement. If you try to READ more data values than exist in the DATA statements, an "Out of data" error message is displayed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

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

This program segment reads the values from the DATA statements into the array A. After execution, the values of A(1) through A(10) are 3.08 through 3.37.

```
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
Ok
```

This program reads string and numeric data from the DATA statement in line 30. Note, quotation marks where needed around "DENVER," because of the comma.

**REM
Statement**

Purpose To allow explanatory remarks to be inserted in a program.

Format REM <remark>

remark may consist of any sequence of characters.

Comments REM statements are not executed but they do take up space in memory. REM statements are output exactly as entered when the program is listed.

REM statements may be branched to from a GOTO or GOSUB statement. Execution continues with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark.

During execution, the interpreter ignores everything on the line after a remark.

Examples

```
120 REM CALCULATE AVERAGE SCORE
130 SUM = 0 : REM INITIALIZE SUM
30 FOR I = 1 TO 20
40 SUM = SUM + SCORE(I) 'ADD SCORES
160 NEXT I
170 'CALCULATE AVERAGE
180 AVERAGE = SUM / 20
```

This example shows ways of adding remarks to your program.

```
10 PRINT "LINE 10"
20 'LINE 20 : PRINT "LINE 20"
30 PRINT "LINE 30"
40 REM LINE 40 : PRINT "LINE 40"
RUN
LINE 10
LINE 30
Ok
```

This example demonstrates the fact that the interpreter ignores everything on a program line after a remark statement.

**RENUM
Command**

Purpose To renumber program lines.

Format RENUM [[<new number>][,<old number>][,<increment>]]

new number is the first line number to be used in the new sequence. The default is 10.

old number is the line in the current program where renumbering is to begin. The default is the first line of the program.

increment is the increment to be used in the new sequence. The default is 10.

Comments The RENUM command is used to renumber the program lines of the program currently in memory. RENUM also changes all line number references following GOTO, GOSUB, IF...THEN...ELSE, ON...GOTO, ON...GOSUB, RESTORE, RESUME, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number xxxxx in yyyy" is displayed. The incorrect line number reference (xxxx) is not changed by RENUM, but line number yyyy may be changed.

Examples RENUM

Renumbers the entire program. The first new line number will be 10. Line numbers are incremented by 10.

RENUM 300,,5

Renumbers the entire program. The first new line number will be 300. Line numbers are incremented by 5.

RENUM 1000,900,20

Renumbers the lines from 900 up so they start with line number 1000 and increments line numbers by 20.

Notes RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error message will be displayed.

**RESET
Command**

Purpose To close all disk files.

Format RESET

Comments RESET closes all open disk files on all drives and writes the directory track to every disk with open files.

The RESET performs the same function as the CLOSE statement with no optional file numbers.

**RESTORE
Statement**

Purpose To allow DATA statements to be reread from a specified line.

Format RESTORE [<line number>]

line number is the line number of a DATA statement in the program.

Comments The RESTORE statement is used to restore the data in DATA statements. Executing a RESTORE statement with no <line number>, resets the pointer used to indicate the next item in the DATA table, back to the first element of the DATA table. The next READ statement would then read the first item in the first DATA statement in the program.

If a <line number> is specified, the DATA table pointer is reset to the beginning of that line. The next READ statement would then read the first data value in the specified DATA statement.

Examples

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
50 PRINT A B C D E F
RUN
 57 68 79 57 68 79
Ok
```

The RESTORE statement in line 20 restores the data in line 40 so the values can be read by line 40.

**RESUME
Statement**

Purpose To continue program execution after an error recovery procedure has been performed.

Format RESUME [0]
 RESUME NEXT
 RESUME <line number>

line number is a valid line number in the program.

Comments The format of the RESUME statement depends upon where you would like execution is to resume:

RESUME [0] Execution resumes at the statement that caused the error.

The 0 is optional, but if you use a RENUM command on a program containing a RESUME 0 statement, an "Undefined line number" error message is displayed.

RESUME NEXT Execution resumes at the statement immediately following the one that caused the error.

RESUME <line number> Execution resumes at <line number>.

A RESUME statement that is not in an error handling routine causes a "RESUME without error" message to be displayed.

Example 10 ON ERROR GOTO 1000
 140 LPRINT
 1000 ***** ERROR HANDLING SUBROUTINES *****
 1010 IF ERR = 27 THEN PRINT "CHECK PRINTER" : GOTO 1100
 1090 ON ERROR GOTO 0
 1000 PRINT "PRESS ANY KEY TO CONTINUE"
 1110 A\$ = ""
 1120 WHILE A\$ = "" : A\$ = INKEY\$: WEND
 1130 RESUME

The error trapping routine in line 1010 is executed if the printer is out of paper or turned off. The loop in line 1120 waits until a key is pressed. When a key is pressed, line 1130 returns the program to the line that caused the error.

**RETURN
Statement**

Purpose To return from a subroutine.

Format RETURN <line number>

 line number is a valid line number in the program.

Comments The RETURN statement is used to transfer control back from a subroutine called by a GOSUB statement. Refer to the GOSUB...RETURN statement for more information on the use of GOSUB and RETURN.

The RETURN <line number> was designed to allow non-local returns from event trapping routines. You will often need to return from an event trapping routine to a specific program segment. The RETURN <line number> format allows you to transfer control to that program segment and eliminate the GOSUB that was created by the event trap. Care should be taken when using non-local returns, since all other GOSUBs, WHILEs, and FORs active at the time of the trap remain active.

Example 40 IF X > 100 THEN GOSUB 300

```

      .
      .
300 'SUBROUTINE
      .
      .
390 RETURN
    
```

This example shows one way of using using the GOSUB...RETURN statements to run a subroutine.

**RIGHT\$
Function**

Purpose To return a string comprised of the rightmost n characters of string x\$.

Format RIGHT\$(X\$,n)

x\$ is any string expression.

n is a numeric expression in the range 0 to 255.

Comments The RIGHT\$ function is used to return a string from the right n characters of string x\$.

If n is equal to the number of characters in x\$ (LEN(X\$)), the entire string (x\$) is returned. If n is zero (0), the null string (length zero) is returned.

Example 10 A\$ = "DISK BASIC"
 20 PRINT RIGHT\$(A\$,5)
 RUN
 BASIC
 Ok

Notes The LEFT\$ and MID\$ functions can also be used to return portions of a string.

**RMDIR
Command**

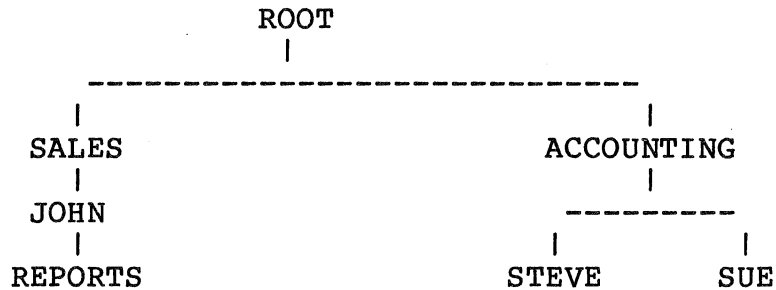
Purpose To remove a directory from the specified disk.

Format RMDIR <path>

path is a string expression indicating the sub-
 directory to be removed from the existing
 directory. <path> cannot exceed 128
 characters.

Comments The directory being removed must be empty of all files
 and sub-directories, except the "." and ".." entries,
 or a "Path/File Access Error" error message is
 displayed.

Example



To remove the directory REPORTS while in the root directory, the following statements could be used:

RMDIR "SALES\JOHN\REPRTS"

or

CHDIR "SALES\JOHN"
RMDIR "REPORTS"

Notes The KILL command cannot be used to remove a directory.

**RND
Function**

Purpose To return a random number between 0 and 1.

Format RND[(x)]

x is a numeric expression that affects the value returned by the RND function.

Comments The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded using the RANDOMIZE statement. The random generator can also be reseeded by using a negative argument. This always generates the same sequence of numbers for any given x, and the sequence is not affected by the RANDOMIZE statement.

If x is positive or omitted, RND(x) generates the next random number in the sequence. If x is zero (0), RND(0) repeats the last number generated.

Example

```
10 PRINT RND(-1) RND
20 PRINT RND(-1) RND RND
30 PRINT RND(0)
RUN
.65086 .6545178
.65086 .6545178 .4623311
.4623311
Ok
```

Line 10 seeds the random number generator with -1, and prints the first two numbers in the sequence. Line 20 reseeds the random number generator again with -1 and prints the first three numbers in the sequence. This shows how a negative argument resets the random number generator to the beginning of the same sequence each time. Line 30 uses the zero (0) argument to print out the last number generated in the sequence.

RUN**Command**

Purpose To execute the program currently in memory.

Format RUN [<line number>]

or

RUN <filespec>[,R]

line number is a line number in the program to start execution at.

filespec is a string expression indicating the file to be loaded and run.

Comments RUN or RUN <line number> starts execution of the program currently in memory. If <line number> is specified, execution begins on that line of the program. Otherwise, execution begins at the first line in the program.

RUN <filespec> loads the specified program from disk into memory and starts execution at the first line number in the program. All open files are closed and the current contents in memory are deleted before the program is loaded. If the ,R option is included, all open data files remain open.

Example RUN

This command begins execution of the program currently in memory, starting at the first line in the program.

RUN "B:NEWFILE"

This command loads the program NEMFILE.BAS from drive B and begins execution at the first line in the program.

**SAVE
Command**

Purpose To save a program file on disk.

Format SAVE <filespec>[,{A|,P}]

filespec is a string expression indicating the file specification for the file to be saved. <filespec> includes an optional drive designation, an optional path, a filename, and an optional filename extension.

Comments If a filename extension is not included, GWBASIC appends the .BAS extension the filename. If a drive designation is not included, the program is saved on the TeleDOS default drive. If a file currently exists on the disk with the same filename, GWBASIC writes the new file over the old file.

The A option saves the file in ASCII format. If the A option is not specified, the file is saved in a compressed binary format. The ASCII format requires more disk space, but some types of access require that the files be in ASCII format. For example, the MERGE command requires the file being merged be in the ASCII format.

The P option protects the file by saving it in an encoded binary format. Any attempt to view the program using the LIST or EDIT commands result in an "Illegal function call" error message being displayed.

The diskette directory entry for a program will not indicate if a file was saved using the ,A or ,P option.

Examples SAVE "MYPROG"

This command saves the program currently in memory on the default drive as MYPROG.BAS.

SAVE "B:COM2",A

This command saves the program currently in memory on drive B as COM2.BAS. The program is saved in the ASCII format.

SAVE "SECRET",P

This command saves the program currently in memory on the default drive as SECRET.BAS. The program is saved in the encoded binary format so it cannot be viewed.

**SCREEN
Function**

Purpose To read a character or its color from a specified screen location.

Format `n = SCREEN(<row>,<column>[,z])`

`n` is an integer in the range 0 to 255.

`row` is a numeric expression in the range 1 to 25 indicating a row on the screen.

`column` is a numeric expression in the range 1 to 40 or 1 to 80, depending upon the screen width, indicating a column position on the screen.

`z` is a numeric expression which evaluates to a true (non-zero) or false (zero) value. The `z` parameter is only valid in the text mode.

Comments The SCREEN function without the optional `z` parameter returns the ASCII code (see Appendix B) of the character at screen position (`row,column`). In the graphics mode, if the specified character position contains graphic information displayed by one of the graphics commands, the SCREEN function returns a value of zero (0).

In the text mode, if the `z` parameter is included and is non-zero, SCREEN returns the color of the character at the specified position. The color is determined from the value `n` as follows:

`foreground = n MOD 16`

`background = ((n - foreground)/16) MOD 128`

If `n` is in the range 128 to 255, the character is blinking.

If values of `<row>` or `<column>` are entered out of the specified range, an "Illegal function call" error message is displayed.

Example `100 X = SCREEN (10,10)`

This statement assigns the ASCII code for the character position (10,10) to the variable `X`. If position (10,10) contained an upper-case `A`, `X` would be set equal to 65.

**SCREEN
Statement**

Purpose To set the specifications for the display screen.

Format SCREEN [<mode>][,<color>][,<active page>]
[,<visual page>]]

mode is a numeric value in the range 0 to 2 indicating the current mode as follows:

0 Text mode at the current width (40 or 80).

1 Medium resolution graphics mode (320 x 200).

2 High resolution graphics mode (640 x 200).

color is a numeric expression resulting in a true (non-zero) or false (zero) value which enables or disables color.

Mode	Value	Color
0	True	Enabled
	False	Disabled
1	True	Disabled
	False	Enabled

active page is a numeric expression in the range 0 to 7 for width 40, and 0 to 3 for width 80. <active page> selects the screen page to be written to by screen output statements. <active page> is only valid in the text mode, mode 0.

visual page is a numeric expression in the range 0 to 7 for width 40, and 0 to 3 for width 80. <visual page> selects the screen page to be displayed on the screen. The <visual page> can be different from the <active page>, by defaults to the <active page> if omitted. <visual page> is only valid in the text mode, mode 0.

Comments When the SCREEN statement is executed with a new screen mode, the new mode is stored, the screen is cleared, the foreground color is set to white, and the background and border colors are set to black.

**SCREEN
Statement**

In the text mode, changing the <active page> and <visual page> parameters changes the page of memory being displayed on the screen. Changing between pages does not affect data written to any of the pages. This allows you to display one page while you are writing to another, then instantly switch to the other page.

If parameters other than <visual page> are omitted, the current value is retained.

Examples SCREEN 0,1,0,0

This statement selects the text mode with color and sets the active and visual page to page 0.

20 SCREEN 0,,1,0

This statement results in all screen output statements to write to page one, while the user still views page 0. The color setting remains unchanged.

50 SCREEN 1,0

This statement selects medium resolution graphics with color.

**SGN
Function**

Purpose To return the sign of a numeric expression.

Format SGN(x)

x is any numeric expression.

Comments The SGN function is the mathematical signum function which returns the following values:

1 if x is greater than 0.

0 if x equals 0.

-1 if x is less than 0.

Example 70 ON SGN(X)+2 GOTO 100,200,300

In this example, program control branches to line 100 if X is negative, 200 if X is 0, and 300 if X is positive.

**SIN
Function**

Purpose To return the trigonometric sine function.

Format SIN(x)

 x is an angle in radians, where pi radians equal 180 degrees.

Comments To convert degrees to radians, multiply degrees by pi/180, where pi = 3.141593.

Example 10 INPUT "ENTER ANGLE IN DEGREES";DEGREES
 20 PI = 3.141593
 30 SINE = SIN(DEGREES * PI / 180)
 40 PRINT "THE SINE OF" DEGREES "DEGREES IS" SINE
 RUN
 ENTER ANGLE IN DEGREES? 90
 THE SINE OF 90 DEGREES IS 1
 Ok

This example program inputs an angle in degrees, converts the angle to radians and finds the sine of the angle, and then prints out the results.

SOUND
Statement

Purpose To generate a sound through the speaker.

Format SOUND <frequency>,<duration>

frequency is a numeric expression in the range 37 to 32767 representing the desired frequency in hertz.

duration is a numeric expression in the range 0 to 65535 indicating the duration in clock ticks. Clock ticks occur 18.2 times per second.

Comments When a SOUND statement is executed, program control continues to the next statement, even though the SOUND <duration> has not been completed. If another SOUND statement is encountered, the program waits until the last sound has been completed. If the duration of the next SOUND statement is zero, the SOUND statement that is running is terminated.

The following table gives a correlation between the notes for two octaves around middle C and their frequencies.

Note	Frequency	Note	Frequency
C	130.810	C*	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

* middle C

To create a pause, or period of silence, a <frequency> of 32767 is used.

**SOUND
Statement**

To determine the <duration> to use for one beat of music, the following formula can be used:

$$\text{one beat} = 1092 / (\text{beats per minute})$$

1092 is the number of clock ticks per minute.

Examples

```
10 FOR I = 1 TO 50
20 SOUND RND*1000+37,2
30 NEXT I
```

This example create random sounds.

```
10 FOR I = 1000 TO 440 STEP -5
20 SOUND I,0.5
30 NEXT I
40 SOUND 50,1
```

This example could be used in a game to imatate the sound of of an object falling.

**SPACE\$
Function**

Purpose To return a string of spaces of a specified length.

Format SPACE\$(x)

x is a numeric expression in the range 0 to 255 indicating the number of spaces in the string. x is rounded to an integer before the string is created.

Comments The SPACE\$ function creates a string consisting of x spaces (ASCII character 32). If x is 0, SPACE\$ returns the null string (length = 0). If x is outside of the specified range, an "illegal function call" error message is displayed.

Example 10 FOR I = 1 TO 5
 20 X\$ = SPACE\$(I)
 30 PRINT X\$;I
 40 NEXT I
 RUN
 1
 2
 3
 4
 5
 Ok

Note Also see SPC function.

**SPC
Function**

Purpose To skip spaces in a PRINT statement.

Format SPC(x)

 x is any numeric expression.

Comments SPC may be used in the PRINT, LPRINT, and PRINT# statements to skip a specified number of spaces, or character positions. The argument x is rounded to an integer to determine the number of spaces to skip. If x is greater than the width of the display, the value x MOD width is used, where width is the width of the screen or device. If x is less than 0 or greater than 32767, no spaces are skipped.

If the SPC function is the last item in a list of data items to be printed, GWBasic does execute a carriage return at the end of the statement. The cursor remains in the character position after the last space skipped.

Example PRINT "OVER" SPC(15) "THERE"
 OVER THERE
 Ok

The SPC function separates the words "OVER" and "THERE" by 15 spaces.

SQR
Function

Purpose To return the square root of the specified argument.

Format SQR(x)

x is a numeric expression greater than or equal to 0.

Comments If x is less than zero, an "Illegal function call" error message is displayed.

Example 10 FOR X=10 TO 25 STEP 5
 20 PRINT X, SQR(X)
 30 NEXT
 RUN
 10 3.162278
 15 3.872984
 20 4.472136
 25 5
 Ok

This example calculates the square roots of the numbers 10, 15, 20, and 25.

**STICK
Function**

Purpose To return the x and y coordinates of the two joysticks.

Format v = STICK(n)

v is a numeric variable to store the result returned by the STICK function.

n is a numeric expression in the range 0 to 3 assigning the following value to v.

0 the x coordinate of joystick A.

1 the y coordinate of joystick A.

2 the x coordinate of joystick B.

3 the y coordinate of joystick B.

STICK(0) actually retrieves all four values and assigns the value to v. Using a value of n from 1 to 3 assigns the values previously retrieved by STICK(0).

Example

```
10 CLS
20 LOCATE 1,1
30 PRINT
40 LOCATE 1,1
50 PRINT STICK(0) ", " STICK(1)
60 GOTO 20
```

This example creates an endless loop to display the value of the x,y coordinate for joystick A in the upper-left corner of the screen.

**STOP
Statement**

Purpose To terminate program execution and return to command level.

Format STOP

Comments STOP statements may be used anywhere in a program to terminate execution. STOP is often used for debugging. When a STOP is encountered, the following message is printed:

Break in line nnnnn

where nnnn indicates the line number of the STOP statement.

The STOP statement does not close files.

Execution of the program can be resumed by issuing a CONT command.

Example 10 INPUT A,B
 20 C = A * B
 30 STOP
 40 D = C^2
 50 PRINT D
 RUN
 ? 2,3
 BREAK IN 30
 Ok
 PRINT C
 6
 Ok
 CONT
 36
 Ok

This example inputs two values, calculates the product of the two, and then stops. While the program is stopped, we can check the value of C. The CONT command resumes execution of the program at line 40.

**STR\$
Function**

Purpose To return a string representation of the specified numeric value.

Format STR\$(x)

x is any numeric expression.

Comments The STR\$ function returns a string made up of the characters in the value represented by the expression x. If x is positive, the string includes a leading space.

Example

```
10 X = 12345
20 X$ = STR$(X)
30 Y$ = "67890"
40 Z = 67890
50 Z$ = STR$(Z)
60 PRINT X$ + Y$ , LEN(X$ + Y$)
70 PRINT X$ + Z$ , LEN(X$ + Y$)
RUN
 1234567890  11
 12345 67890  12
Ok
```

This example shows how the STR\$ function places a leading space in the string returned from a positive number.

Notes The VAL function is the complement of the STR\$ function, and returns the numeric value of a string.

STRIG
Function and Statement

 Purpose To return the status of a specified joystick button.

Format STRIG ON statement
 STRIG OFF

STRIG(n) function

n is a number in the range 0 to 7 which determines the value returned by the STRIG function as follows:

- 0 Returns -1 if trigger A1 was pressed since the last STRIG(0) statement, returns 0 if not.
- 1 Returns -1 if trigger A1 is currently down, returns 0 if not.
- 2 Returns -1 if trigger B1 was pressed since the last STRIG(2) statement, returns 0 if not.
- 3 Returns -1 if trigger B1 is currently down, returns 0 if not.
- 4 Returns -1 if trigger A2 was pressed since the last STRIG(4) statement, returns 0 if not.
- 5 Returns -1 if trigger A2 is currently down, returns 0 if not.
- 6 Returns -1 if trigger B2 was pressed since the last STRIG(6) statement, returns 0 if not.
- 7 Returns -1 if trigger B2 is currently down, returns 0 if not.

Comments The STRIG ON statement must be executed before any STRIG(n) function calls can be made. After a STRIG ON, GWBASIC checks between execution of each statement to see if a button has been pressed.

If a STRIG OFF statement is executed, or before a STRIG ON statement is executed, no testing takes place.

STRIG**Function and Statement**

Example 120 STRIG ON
 130 A = STRIG(0) : B = STRIG(2)
 140 WHILE A = 0 AND B = 0
 150 A = STRIG(0)
 160 B = STRIG(2)
 170 WEND
 180 IF A = -1 THEN GOSUB 250
 190 IF B = -1 THEN GOSUB 500
 200 GOTO 130

In this example, if button A1 or B1 has not been pressed since the last time the buttons were checked, the program enters the WHILE...WEND loop and waits until one of the buttons is pressed. When a button is pressed, the program goes and executes the proper subroutine and then returns to the loop.

**STRIG(n)
Statement**

Purpose To enable and disable joystick event trapping.

Format STRIG(n) ON

STRIG(n) OFF

STRIG(n) STOP

n may be 0, 2, 4, or 6, indicating the button to be trapped as follows:

0	button A1
2	button B1
4	button A2
6	button B2

Comments The STRIG(n) ON statement enables joystick event trapping by an ON STRIG(n) statement. After a STRIG(n) ON statement, and if a non-zero line number was entered in the ON STRIG(n) statement, GWBASIC checks between execution of each statement to see if the specified joystick button has been pressed.

The STRIG(n) OFF statement disables event trapping. If an event occurs, it will not be remembered.

The STRIG(n) STOP statement disables event trapping, but if an event occurs it will be remembered. When the next STRIG(n) ON statement is executed, the event trap takes place, and the GOSUB portion of the ON STRIG(n) statement is executed.

Example

```
10 ON STRIG(0) GOSUB 600
20 ON STRIG(2) GOSUB 800
30 STRIG(0) ON
40 STRIG(2) ON
```

This program segment sets up the event trapping for joystick buttons A1 and B1.

**STRING\$
Function**

Purpose To return a string of length n whose characters all have ASCII code m or the first character of x\$.

Format STRING\$(n,m)

 STRING\$(n,x\$)

n is a numeric expression in the range 0 to 255 indicating the length of the string.

m is a numeric expression in the range 0 to 255 indicating the ASCII code of the character contained in the string.

x\$ is any string expression.

Comments If values of n and m are entered outside of the specified range, an "Illegal function call" error message is displayed.

Examples 10 X\$ = STRING\$(10,45)
 20 PRINT X\$ "MONTHLY REPORT" X\$
 RUN
 -----MONTHLY REPORT-----
 Ok

This example uses the STRING\$ function to print a string of ten hyphens, ASCII character 45.

```
10 X$ = "HELLO"
20 PRINT STRING$(5,X$)
RUN
HHHHH
Ok
```

In this example, the STRING\$ functin returns a string comprised of the letter H, the first letter of the word HELLO.

**SWAP
Statement**

Purpose To exchange the values of two variables.

Format SWAP <variable>,<variable>

 variable is any valid variable type.

Comments The SWAP statement swaps, or exchanges the values assigned to two variables. Any type of variable may be used with the SWAP statement (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error message is displayed.

 If the second variable entered is not defined when the SWAP statement is executed, an "Illegal function call" error message is displayed.

Example 10 A\$ = " ONE " : B\$ = "FOR" : C\$ = " ALL "
 20 PRINT A\$ B\$ C\$
 30 SWAP A\$, C\$
 40 PRINT A\$ B\$ C\$
 RUN
 ONE FOR ALL
 ALL FOR ONE
 Ok

 This example uses the SWAP statement to swap the string values of strings A\$ and C\$.

**SYSTEM
Command**

Purpose To close all open files and return control to the
operating system.

Format SYSTEM

Comments When a SYSTEM command is executed, GWBASIC closes any
files that are open, and then returns you to the
TeleDOS operating system.

If GWBASIC was entered from a batch file, the SYSTEM
command returns control to the batch file.

**TAB
Function**

Purpose To tab to a specified position.

Format TAB(n)

n is a numeric expression indicating the character position to tab to.

Comments The TAB function is used with the PRINT, LPRINT, and PRINT# statements to indicate the position of the next character to be printed. If the current print position is already beyond character position n, TAB goes to that position on the next line.

TAB position 1 is the first character position on the line. The last position on the line is equal to the width of the screen (40 or 80). If n is greater than width, TAB uses the value (n MOD width) as the character position to TAB to. If n is 0, negative, or greater than 32767, a TAB(1) is performed.

Example

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES", "$25.00"
RUN
NAME                AMOUNT
G. T. JONES         $25.00
```

Notes When the TAB function is used in a PRINT statement, spaces are printed out to the specified character position. To preserve items already printed on the line, use the LOCATE statement and then the PRINT statement.

**TAN
Function**

Purpose To return the trigonometric tangent value of a specified argument.

Format TAN(x)

x is an angle in radians. To convert degrees to radians, multiply degrees by pi/180 (where pi = 3.141593).

Comments The TAN function is performed in single-precision. If the /D option was included when GWBASIC was started, TAN is performed in double-precision.

Example 10 PI = 3.141593
 20 INPUT "ENTER AN ANGLE IN DEGREES";DEGREES
 30 PRINT "THE TANGENT OF" DEGREES "DEGREES IS";
 40 PRINT TAN(DEGREES * PI / 180)
 RUN
 ENTER AN ANGLE IN DEGREES? 45
 THE TANGENT OF 45 DEGREES IS 1
 Ok

This program inputs an angle in degrees and then prints out the tangent of the angle.

**TIME\$
Function**

Purpose To retrieve the current time known to the system.

Format TIME\$

Comments The TIME\$ function returns an eight-character string in the form:

hh:mm:ss

hh is the current hour (00 through 23)

mm is the current minute (00 through 59)

ss is the current second (00 through 59)

A 24-hour clock is used; 8:00 p.m. would be shown as 20:00:00.

The current time can be set by the TIME\$ statement, or in TeleDOS before entering GWBasic.

Example PRINT TIME\$
08:55:03
Ok

**TIME\$
Statement**

Purpose To set the time known to the system.

Format TIME\$ = <string expression>

string expression is a string expression indicating the current time. The three following forms can be used:

hh Set the current hour, where hh is in the range 0 to 23. The minutes and seconds default to 00.

hh:mm Sets the current hour and minute. Minutes must be in the range 0 to 59. Seconds default to 00.

hh:mm:ss Sets the current hour, minute, and second. Seconds must be in the range 0 to 59.

Comments A 24-hour clock is used; 8:00 p.m. would be entered as 20:00:00.

Leading zeros may be omitted from any of the entires, but you must enter at least one digit.

If a value for hh, mm, or ss is entered out of the specified range, an "Illegal function call" error message is displayed and the current time is not changed.

Example TIME\$ = "08:00:00"

This statement sets the current time to 8:00 a.m.

**TIMER
Function**

Purpose To return a single-precision number representing the elapsed number of seconds since midnight or the last system reset (startup or <Ctrl>/<Alt>/).

Format TIMER

Comments The TIMER function is a read only function which returns the number of seconds that have elapsed since midnight, or since the system was started or reset.

Examples 30 RANDOMIZE TIMER

This program line reseeds the random number generator with a unique number almost every time the program is run.

**TRON/TROFF
STATEMENTS/COMMANDS**

Purpose To trace the execution of program statements.

Format TRON
TROFF

Comments As an aid in debugging, the TRON statement (which can be entered in either direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example

```

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
TRON
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
    
```

This example shows the form of the output when the TRACE is on.

**USR
Function**

Purpose To call an assembly language subroutine.

Format USR[n][(<argument>)]

n is a number in the range 0 to 9 specifying which USR routine is being called. If n is omitted, USR0 is assumed.

argument is a numeric or string expression being passed to the subroutine as an argument.

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

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed.

Example 100 DEF SEG=&H8000
110 DEF USR0=0
120 X = 5
130 Y = USR0(X)

This example set the current segment to hex 80000 and the offset of the subroutine to 0. Line 130 calls the subroutine and passes the argument X.

**VAL
Function**

Purpose To return the numerical value of a specified string.

Format VAL(x\$)

 x\$ is any valid string expression.

Comments The VAL function strips leading blanks, tabs, and linefeeds from the argument string and returns the value of the numerical characters at the beginning of the string. If the first valid character is not numeric, VAL returns a value of zero (0).

Example 10 X1\$ = " -34"
 20 X2\$ = "1170 MORSE"
 30 X3\$ = "SUNNYVALE, CA"
 40 PRINT VAL(X1\$) VAL(X2\$) VAL(X3\$)
 RUN
 -34 1170 0
 Ok

Notes The STR\$ function is the complement of the VAL function and returns a string expression from a numeric expression.

**VARPTR
Function**

Purpose To return the address in memory of the specified variable or file control block.

Format VARPTR(<variable>)

VARPTR(#<file number>)

variable is the name of a string or numeric variable or array element in your program.

file number is the number under which the file was opened in the OPEN statement.

Comments The VARPTR function returns an integer value in the range 0 to 65535, indicating the offset into GWBasic's data segment. The address is not affected by a DEF SEG statement.

For the VARPTR(<variable>) format, the value returned points to the data portion of the stored variable (refer to Appendix ?? for an explanation on how variables are stored). A value must be assigned to <variable> prior to execution of VARPTR or an "Illegal function call" error message is displayed.

You should assign values to all the simple variables in your program before you execute the VARPTR function for an array. The addresses of arrays change whenever new simple variables are assigned.

VARPTR is usually used to obtain the address of a variable or array so that it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

For the VARPTR(#<file number>) format, the value returned is the starting address for the GWBasic file control block for the file. This file control block is different from the TeleDOS file control block. Refer to Appendix ?? for information on the format of the GWBasic file control block.

Example 100 X=USR0(VARPTR(Y))

This example passes the starting address of the data for variable Y to the assembly language subroutine USR0.

**VARPTR
Function**

```
130 CHK = PEEK(VARPTR(#1))  
140 ON CHK GOTO 200,300
```

This example checks to see which mode file number one is open in. If it is open in the Input mode, control is transferred to line 200, if it is open in the Output mode, control is transferred to line 300.

VARPTR\$
Function

Purpose To return a character string form of the memory address of the variable.

Format VARPTR\$(*<variable>*)

variable is the name of a variable in your program.
<variable> can be string, numeric, or an array element.

Comments A value must be assigned to *<variable>* prior to execution of VARPTR\$ or an "Illegal function call" error message is displayed.

VARPTR\$ returns a three-byte string in the form:

byte 0 = type
 byte 1 = low byte of address
 byte 2 = high byte of address

where type indicates the variable type as follows:

2 integer
 3 string
 4 single-precision
 8 double-precision

The characters in the string are the ASCII characters represented by the numeric value in those data bytes. For example, if you printed the string returned for a single-precision variable, the first character in the string would be a diamond, which is the ASCII character 4.

Example 10 A = 24
 20 PRINT VARPTR(A)
 30 X\$ = VALPTR\$(A)
 30 PRINT 256 * ASC(MID\$(X\$,3,1) + ASC(MID\$(X\$,2,1))
 RUN
 3398
 3398
 Ok

Note Because array addresses change whenever a new simple variable is assigned, always assign all simple variables before calling VARPTR\$ for an array element.

VIEW**Statement**

Purpose To allow you to define a subset of the screen, called a viewport, onto which WINDOW contents are mapped. For use in graphics mode only.

Format VIEW [[SCREEN] [(x1,y1)-(x2,y2)[,[<color>] [, [<border>]]]]]

(x1,y1)-(x2,y2) are the upper-left and lower-right coordinates or the viewport being defined. The x and y coordinates entered must be within the limits of the screen (x in the range 0 to 319 in medium resolution and 0 to 639 in high resolution; y in the range 0 to 199) or an "Illegal function call" error message is displayed.

color allows you to fill the specified viewport in color <color>. If <color> is omitted, the viewport is not filled. <color> can range from 0 to 3 in medium resolution, where 0 indicates the background color, and 1 to 3 indicate the colors of the current palette. In high resolution, <color> can be 0 for black or 1 for white.

border allows you to draw a border line around the specified viewport. The border is drawn on the points just outside of the viewport. If the viewport includes points along the edge of the screen, a border is not drawn outside of these points. <border> indicates the color the border is to be drawn in, using the colors as described in <color>.

Comments VIEW sorts the coordinates in the x and y pairs, using the smaller values as x1 and y1. For example, the statement:

```
VIEW (100,5)-(5,100)
```

is converted to:

```
VIEW (5,5)-(100,100)
```

**VIEW
Statement**

If x1 equals x2, or y1 equals y2, an "Illegal function call" error message is displayed.

When the SCREEN argument is omitted, all graphics points are plotted relative to the upper-left corner of the viewport. This means the values of x1 and y1 are added to the x and y coordinates before the points are plotted on the screen. For example, if

```
VIEW(10,10)-(200,100)
```

is executed, then the point plotted by the statement PSET (0,0),3 is plotted at screen coordinate (10,10).

When the SCREEN argument is included, all graphics points plotted are absolute, that is, relative to the center of the current coordinate system.

In either format, points with coordinates outside of the viewport are not displayed. For example, if

```
VIEW SCREEN (10,10)-(200,100)
```

is executed, then the point plotted by the statement PSET (0,0),3 would not be displayed because it is outside of the specified viewport.

The VIEW statement with no arguments defines the entire screen as the current viewport. RUN and SCREEN disable any current viewport and reset the viewport to the entire screen.

The VIEW statement can be used with the WINDOW statement to do scaling. Since the points of the current window are mapped into the current viewport, changing the size of the viewport changes the size of the object being drawn.

```
Example 10 SCREEN 1,0 : CLS
        20 WINDOW SCREEN (0,0)-(319,199)
        30 GOSUB 80 'NORMAL SIZE
        40 FOR I = 1 TO 100 : NEXT I : CLS
        50 VIEW (40,40)-(289,159),,3
        60 GOSUB 80 'REDUCED SIZE
        70 END
        80 '*** CIRCLES ***
        90 CIRCLE(160,100),60,3,,,5/20
        100 CIRCLE(160,100),60,3,,,1
        110 RETURN
```

**VIEW
Statement**

In this example, the VIEW statement is used to reduce the set of circles drawn using the CIRCLE statements in lines 90 and 100. Line 20 maps the screen into the full screen the first time the circles are drawn. The second time they are drawn, line 50 maps them into the smaller viewport.

Notes When using VIEW, the CLS statement only clears the current viewport. The CLS statement does not HOME the cursor if the current viewport is not the entire screen.

VIEW PRINT

Statement

 Purpose To set the boundaries of the text window.

Format VIEW PRINT [<top line> TO <bottom line>]

top line is a numeric expression in the range 1 to 25 indicating the top screen line to be used by the text window.

bottom line is a numeric expression in the range 1 to 25 indicating the bottom screen line to be used by the text window.

Comments The VIEW PRINT statement allows you to limit which screen lines are used as the text window. The Text Editor will limit functions such as scrolling and cursor movement to the designated text window.

Display line 25 can only be included in the text window if the soft key display has been turned off using the KEY OFF statement.

If the VIEW PRINT statement is entered with no parameters, the text window includes the whole screen.

Examples VIEW PRINT 5 TO 15

This statement limits the text window on the screen to lines 5 through 15. To return the text window to the full screen, enter the command:

VIEW PRINT

WAIT
Statement

Purpose To suspend program execution while monitoring the status of a machine input port.

Format WAIT <port>n[,m]

port is a numeric expression in the range 0 to 65535 representing the port number.

n and m are integer expressions in the range 0 to 255.

Comments The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is XOR'ed with the integer expression m, and then AND'ed with n. If the result is zero, GWBASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If m is omitted, it is assumed to be zero.

The WAIT statement allows you to test one or more bit positions in a byte returned by an input port. To test a bit position for a 1, place a one in the same bit position of n, and a 0 in m. To test a bit position for a 0, place a 1 in the same bit position of both n and m.

Bit position	7	6	5	4	3	2	1	0
								1
							2	
Value to place						4		
in n or m					8			
				16				
			32					
		64						
	128							

To check several bits, add the values for the bit positions from the table.

Example 100 WAIT 32,2

This statement suspends program execution until port 32 returns a byte with a 1 in the second bit position.

Notes It is possible to enter an infinite loop with the WAIT statement. A <Ctrl>/<Break> or system reset (<Ctrl>/<Alt>/) can be used to exit the loop.

WHILE...WEND
Statements

Purpose To execute a series of statements in a loop as long as a given condition is true.

Format WHILE <expression>
.
.
[<loop statements>]
.
.
WEND

expression is an expression returning a true (non-zero) or false (zero) value.

Comments If <expression> is true, the following loop statements are executed until the WEND statement is encountered. GWBASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

If <expression> is false when the WHILE statement is first encountered, program control is transferred to the first statement after the WEND statement and the loop statements are not executed.

WHILE...WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement results in a "WHILE without WEND" error message being displayed, and an unmatched WEND statement results in a "WEND without WHILE" error message being displayed.

Example

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
120   FLIPS=0
130   FOR I=1 TO J-1
140     IF A$(I) > A$(I+1) THEN
150       SWAP A$(I),A$(I+1) : FLIPS = 1
160     NEXT I
160 WEND
```

This program segment sorts the elements of array A\$ into alphabetical order using a bubble sort. A\$ was defined with J elements. The WHILE...WEND loop is repeated as long as at least one swap is made.

WIDTH
Statement

Purpose To set the printed line width in number of characters for the screen or line printer.

Format WIDTH <size>
WIDTH #<file number>,<size>
WIDTH <device>,<size>

size is a numeric expression in the range 0 to 255 specifying the new width. WIDTH 0 is equal to WIDTH 1.

file number is a numeric expression in the range 1 to 15 indicating the number of a file opened to one of the valid devices.

device is a string expression indicating the device that is to be used. Valid devices are SCRN:, LPT1:, LPT2:, LPT3:, COM1:, and COM2:.

Comments WIDTH or WIDTH "SCRN:",<size> sets the width of the display screen. Only a <size> of 40 or 80 are recognized. If the WIDTH statement changes the width, the screen is cleared and the border is set to black.

If the screen is in medium resolution graphics, a WIDTH 80 statement changes the screen to the high resolution graphics mode. If the screen is in the high resolution graphics mode, a WIDTH 40 statement changes the screen to the medium resolution graphics mode.

The WIDTH <file number>,<size> statement immediately changes the width of the device associated with <file number> to the specified size. This allows the width to be changed while the file is open.

The WIDTH <device>,<size> statement stores the new width assignment for the specified <device>, but the current setting is not changed. A subsequent OPEN statement for the <device> will use the new width value for the width while the file is open.

If a <size> value of 255 is entered, line folding is disabled; this means a carriage return is not inserted after 255 characters. This has the effect of having an infinite line width. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255. WIDTH 255 is the default value for communications files.

WIDTH
Statement

Changing the width for a communications file does not affect the size of the receive or transmit buffers. The WIDTH statement causes GWBasic to insert a carriage return after every <size> number of characters.

If a value is entered outside of the specified ranges, an "Illegal function call" error message is displayed.

```
Example  10 SCREEN 1,0 'SET TO MED RES COLOR GRAPHICS
          .
          .
          .
          450 SCREEN 0,1 'RETURN TO COLOR TEXT MODE
          460 WIDTH 80 'SET TEXT MODE WIDTH TO 80
          470 END
```

Line 10 of this example sets the screen mode to medium resolution color graphics. Line 450 resets the screen mode to color text mode. Because changing from medium resolution graphics to the text mode leaves you in the 40-column text mode, line 460 is used to set the screen to the 80-column text mode.

WINDOW
Statement

Purpose To allow you to redefine the coordinates of the screen.

Format WINDOW [[SCREEN] (x1,y1)-(x2,y2)]

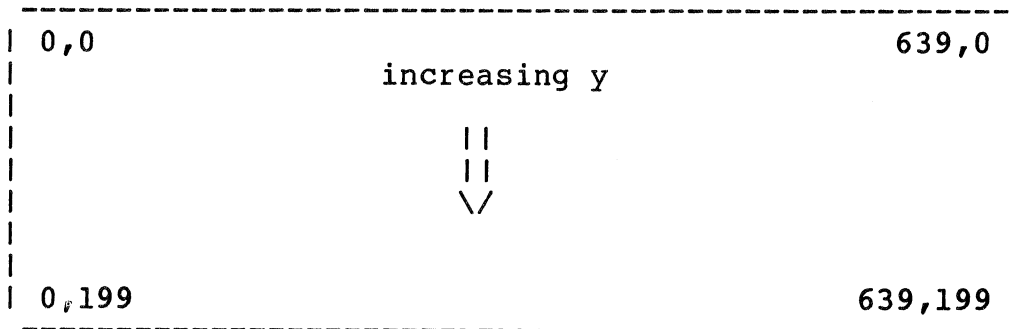
(x1,y1)-(x2,y2) are single-precision floating point numbers defining the coordinates in real space that will be mapped into the viewport defined by the VIEW statement.

Comments The WINDOW statement allows you to define what portion of the Cartesian coordinate system in real space to map into the portion of the screen defined by the VIEW statement.

Normally you view the entire screen as the viewport.
The statements:

```
NEW
SCREEN 2
```

map the points within the rectangle (0,0), (639,0), (639,199), (0,199) into the screen as follows:

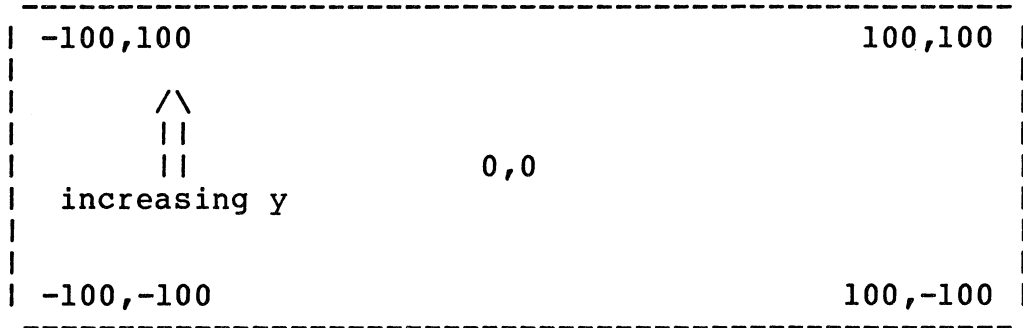


The WINDOW statement allows you to redefine the points on the screen and how there displayed. The statement:

```
WINDOW (-100,-100)-(100,100)
```

would map the points from the rectangle (-100,-100), (100,-100), (100,100), (-100,100) into the screen as follows:

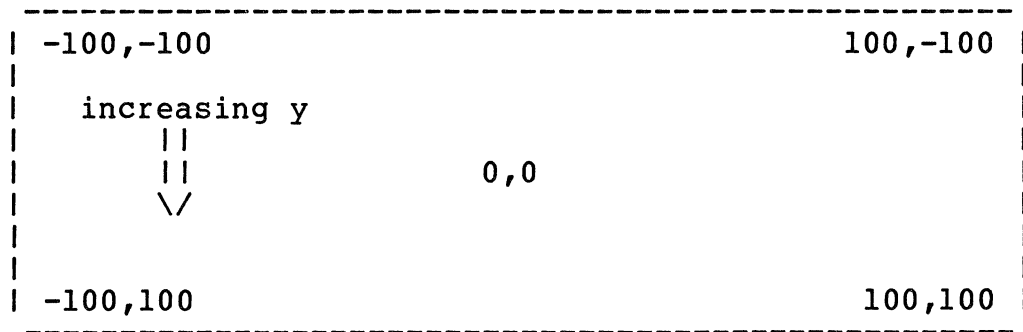
WINDOW
Statement



The direction of increasing y is inverted from the normal graphics screen mode. The addition of the SCREEN argument:

```
WINDOW SCREEN (-100,-100)-(100,100)
```

would map the same rectangle into the screen as follows:



WINDOW sorts the values in the x and y pairs and uses the smaller values for x1 and y1. For example, the statement:

```
WINDOW (100,-50)-(-50,50)
```

becomes:

```
WINDOW (-50,-50)-(100,50)
```

Any possible pairing of x and y are valid, as long as x1 does not equal x2, and y1 does not equal y2. In this case, an "Illegal function call" error message is displayed.

**WINDOW
Statement**

The WINDOW statement allows you to draw anywhere in the Cartesian coordinate space and then select a portion, or window, of this space to look at. Points drawn outside of the designated window are clipped, meaning they are not displayed.

RUN, SCREEN, and WINDOW with no arguments disables any current WINDOW coordinates and return you to the normal screen coordinates.

```
Example 10 KEY OFF : SCREEN 1 : CLS
        20 FOR S = 300 TO 50 STEP -50
        30 WINDOW (-S,-S)-(S,S)
        40 GOSUB 100
        50 NEXT S
        60 END
        100 '*** DRAW GRID & BOX ***
        110 CLS
        120 LINE (-300,0)-(300,0),1 'X AXIS
        130 LINE (0,-300)-(0,300),1 'Y AXIS
        140 'TICK MARKS
        150 LINE (-5,50)-(5,50),2
        160 LINE (-5,-50)-(5,-50),2
        170 LINE (-5,100)-(5,100),2
        180 LINE (-5,-100)-(5,-100),2
        190 LINE (50,-5)-(50,5),2
        200 LINE (-50,-5)-(-50,5),2
        210 LINE (100,-5)-(100,5),2
        220 LINE (-100,-5)-(-100,5),2
        230 ' DRAW BOX
        240 LINE (-60,-40)-(60,40),3,B
        250 FOR I = 1 TO 1000 : NEXT I 'DELAY
        260 RETURN
```

This example show the effect of changing the window to zoom in on the object being drawn. This example also shows how the WINDOW statement clips points being drawn outside of the viewport.

**WRITE
Statement**

Purpose To output data to the screen.

Format WRITE [<expression>[j,j]<expression>...]

expression is a numeric or string expression.
<expression>s can be separated by commas
or semicolons.

Comments If no <expression>s are listed, a blank line is output.

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, GWBASIC inserts a carriage return/linefeed.

The WRITE statement is similar to the PRINT statement, with the following exceptions:

- * the WRITE statement separates the displayed items with commas.
- * strings are delimited with quotation marks.
- * numeric values are not printed with a trailing space.
- * positive numeric values are not preceded by a space.
- * a space cannot be used as a delimiter between the <expression>s to be printed.

Example 10 A = 80 : B = 90 : C\$ = "THAT'S ALL"
20 WRITE A,B;C\$
RUN
80,90,"THAT'S ALL"
Ok

This example shows how the WRITE statement displays numeric and string values.

Notes The TAB(n) and SPC(n) cannot be used with the WRITE statement.

**WRITE#
Statement**

Purpose To write data to a sequential file.

Format WRITE#<file number>,<expression>[,{,|;}<expression>...]

file number is the number under which the file was opened for output in the OPEN statement.

<expression> is a string or numeric expression to be written to file. <expression>s can be separated by commas or semicolons.

Comments The differences between WRITE# and PRINT# statements are:

- * WRITE# inserts commas between the items as they are written to the file.
- * WRITE# delimits strings with quotation marks.
- * WRITE# does not add a trailing space with numeric values or a preceding space with positive numbers.

With the WRITE# statement it is not necessary to put explicit delimiters in the list.

WRITE# also writes a carriage return/linefeed sequence to the file after the last item in the list of <expression>s is written.

Example If A\$ = "CAMERA" and B\$ = "93604-1", the statement:

WRITE#1,A\$,B\$

would write the following image to disk:

"CAMERA","93604-1"

If the following INPUT# statement were executed:

INPUT#1,A\$,B\$

"CAMERA" would be assigned to A\$ and "93604-1" would be assigned to B\$.

APPENDIX A ERROR CODES AND ERROR MESSAGES

GWBASIC has been designed to display an **error message** if an error has been detected that causes a program to stop running. Using the ON ERROR Statement and the variables ERR and ERL, GWBASIC can trap and test the error. (Chapter 5, **GWBASIC Commands, Statements, and Functions**, gives complete explanations of ON ERROR, ERR, and ERL.)

Appendix A has been divided into two tables. Table A-1 is a quick reference guide of error messages and their corresponding number. The second table lists the GWBASIC error messages in numerical order including an explanation of the message.

**Table A-1
Error Message Quick Reference Guide**

#	Message	#	Message
1	NEXT without FOR	29	WHILE without WEND
2	Syntax error	30	WEND without WHILE
3	RETURN without GOSUB	50	FIELD overflow
4	Out of data	51	Internal error
5	Illegal function call	52	Bad file number
6	Overflow	53	File not found
7	Out of memory	54	Bad file mode
8	Undefined line number	55	File already open
9	Subscript out of range	57	Device I/O error
10	Duplicate Definition	58	File already exists
11	Division by zero	61	Disk full
12	Illegal direct	62	Input past end
13	Type mismatch	63	Bad record number
14	Out of string space	64	Bad file name
15	String too long	66	Direct statement in file
16	String formula too complex	67	Too many files
17	Can't continue	68	Device unavailable
18	Undefined user function	69	Communication buffer overflow
19	No RESUME		
20	RESUME without error	70	Disk Write Protect
22	Missing operand	71	Disk not ready
23	Line buffer overflow	72	Disk media error
24	Device Timeout	74	Rename across disks
25	Device Fault	75	Path/file access error
26	FOR without NEXT	76	Path not found
27	Out of paper		

Table A-2
Error Messages

Number	Message
1	<p>NEXT without FOR</p> <p>A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.</p>
2	<p>Syntax error</p> <p>A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).</p>
3	<p>Return without GOSUB</p> <p>A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.</p>
4	<p>Out of data</p> <p>A READ statement is executed when there are no DATA statements with unread data remaining in the program.</p>
5	<p>Illegal function call</p> <p>A parameter that is out of range is passed to a math or string function. This error may also occur as the result of:</p> <ol style="list-style-type: none">1. A negative or unreasonably large subscript.2. A negative or zero argument with LOG.3. A negative argument to SQR.4. A negative mantissa with a noninteger exponent.5. A call to aUSR function for which the starting address has not yet been given.6. An improper argument to MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or ON...GOTO.7. A negative record number used with GET or PUT.

6 Overflow

The result of a calculation is too large to be represented in GWBASIC number format. If underflow occurs, the result is zero and execution continues without an error.

7 Out of memory

A program is too large, has too many FOR loops or GOSUBS, too many variables, expressions that are too complicated or complex PAINTing is used.

8 Undefined line

A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.

9 Subscript out of range

An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.

10 Duplicate definition

Two DIM statements are given for the same array; or, a DIM statement is given for an array after the default dimension of 10 has been established for that array.

11 Division by zero

A division by zero is encountered in an expression or you tried to raise zero to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the exponentiation, 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 BASIC to exceed the amount of free memory remaining. GWBASIC allocates string space dynamically, until it runs out of memory.

- 15 String too long
An attempt is made to create a string more than 255 characters long.
- 16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.
- 17 Can't continue
An attempt is made to continue a program that:
1. Has halted due to an error.
 2. Has been modified during a break in execution.
 3. Does not exist.
- 18 Undefined user function
A user defined function is called before the function definition (DEF FN statement) is given.
- 19 No RESUME
An error handling routine is entered but contains no RESUME statement.
- 20 RESUME without error
A RESUME statement is encountered before an error handling routine is entered.
- 21 Unprintable error
An error message is not available for the error condition that exists.
- 22 Missing operand
An expression contains an operator with no operand following it.
- 23 Line buffer overflow
An attempt has been made to input a line that has too many characters.

- 24 Device timeout
- The device you have specified is not available at this time. This occurs when GWBASIC does not receive information back from an I/O device within a predetermined amount of time.
- 25 Device fault
- An incorrect device designation has been entered.
- 26 FOR without NEXT
- A FOR statement was encountered without a matching NEXT.
- 27 Out of paper
- The printer device is out of paper or not turned on.
- 28 Unprintable error
- An error message is not available for the condition which exists.
- 29 WHILE without WEND
- A WHILE statement does not have a matching WEND.
- 30 WEND without WHILE
- A WEND statement was encountered without a matching WHILE.
- 31-49 Unprintable error
- An error message is not available for the condition which exists.
- 50 Field overflow
- A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
- 51 Internal error
- An internal malfunction has occurred in GWBASIC. Report to computer dealer the conditions under which the message appeared.

- 52 Bad file number
- A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
- 53 File not found
- A LOAD, KILL, NAME, FILES, or OPEN references a file that does not exist on the specified disk.
- 54 Bad file mode
- An attempt is made to use PUT, GET, or LOF with a sequential file or a closed file, to MERGE a non-ASCII file, or to execute an OPEN statement with a file mode other than I, O, or R.
- 55 File already open
- A sequential output mode OPEN statement is issued for a file that is already open; or a KILL statement is given for a file that is open.
- 56 Unprintable error
- An error message is not available for the condition that exists.
- 57 Device I/O error
- An I/O error occurred on a disk I/O operation. TeleDOS cannot recover from the error.
- 58 File already exists
- The filename specified in a NAME statement is identical to a filename already in use on the disk.
- 59-60 Unprintable error
- An error message is not available for the condition that exists.
- 61 Disk full
- All disk storage space is in use. Files are closed when this error occurs.
- 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 (32,767) or equal to zero.
- 64 Bad file name
- An illegal form is used for the filename with a BLOAD, BSAVE, FILES, NAMES, LOAD, SAVE, KILL, or OPEN statement (e.g., a filename with too many characters).
- 65 Unprintable error
- An error message is not available for the condition that exists.
- 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.
- 68 Device Unavailable
- The device that has been specified is not available at this time.
- 69 Communications buffer overflow
- Not enough buffer space has been reserved for communications I/O.
- 70 Disk write protected
- You tried to write to a write protected diskette.
- 71 Disk not ready
- The disk drive door is open or the disk drive does not contain a diskette.
- 72 Disk media error
- A hardware or disk problem occurred while the disk was being written to or read from. For example, the disk may be damaged or the disk drive may not be working properly.

74 Rename across disks

An attempt was made to rename a file with a new drive designation. This is not allowed.

75 Path/file access error

An attempt was made, during an OPEN, NAME, MKDIR, CHDIR, or RMDIR, to use a filename or path to an inaccessible file.

76 Path not found

TeleDOS is unable to find the path the way it is specified during an OPEN, MKDIR, CHDIR or RMDIR.

APPENDIX B ASCII CHARACTER CODES

Table B-1 lists, in decimal, all of the ASCII codes and their associated characters. ASCII codes 0 to 31 have an extra column listing their standard communications functions. The ASCII code characters not corresponding to keys on the keyboard, can be displayed using the PRINT CHR\$(n) function, where n is the ASCII code for the character.

**Table B-1
ASCII Character Codes**

ASCII value	Character	Control character	ASCII value	Character
000	(null)	NUL	032	(space)
001	☺	SOH	033	!
002	☻	STX	034	"
003	♥	ETX	035	#
004	♦	EOT	036	\$
005	♣	ENQ	037	%
006	♠	ACK	038	&
007	(beep)	BEL	039	'
008	▣	BS	040	(
009	(tab)	HT	041)
010	(line feed)	LF	042	*
011	(home)	VT	043	+
012	(form feed)	FF	044	,
013	(carriage return)	CR	045	-
014	🎵	SO	046	.
015	☼	SI	047	/
016	▶	DLE	048	0
017	◀	DC1	049	1
018	↕	DC2	050	2
019	!!	DC3	051	3
020	⌘	DC4	052	4
021	§	NAK	053	5
022	▬	SYN	054	6
023	↕	ETB	055	7
024	↑	CAN	056	8
025	↓	EM	057	9
026	→	SUB	058	:
027	←	ESC	059	;
028	(cursor right)	FS	060	<
029	(cursor left)	GS	061	=
030	(cursor up)	RS	062	>
031	(cursor down)	US	063	?
			064	@
			065	A
			066	B
			067	C
			068	D
			069	E
			070	F

ASCII value	Character	ASCII value	Character	ASCII value	Character
071	G	110	n	149	ò
072	H	111	o	150	ô
073	I	112	p	151	ù
074	J	113	q	152	ÿ
075	K	114	r	153	Ö
076	L	115	s	154	Ü
077	M	116	t	155	€
078	N	117	u	156	£
079	O	118	v	157	¥
080	P	119	w	158	Pt
081	Q	120	x	159	f
082	R	121	y	160	á
083	S	122	z	161	í
084	T	123	{	162	ó
085	U	124		163	ú
086	V	125	}	164	ñ
087	W	126	~	165	Ñ
088	X	127	☐	166	à
089	Y	128	Ç	167	ç
090	Z	129	ü	168	ˆ
091	[130	é	169	⌋
092	\	131	â	170	⌋
093]	132	ä	171	½
094	^	133	à	172	¼
095	_	134	â	173	ı
096	,	135	ç	174	«
097	a	136	ê	175	»
098	b	137	ë	176	•
099	c	138	è	177	☐
100	d	139	ï	178	☐
101	e	140	î	179	
102	f	141	ı	180	⊥
103	g	142	Ä	181	≡
104	h	143	Å	182	≡
105	i	144	É	183	⌋
106	j	145	æ	184	≡
107	k	146	Æ	185	≡
108	l	147	ô	186	≡
109	m	148	ö	187	≡

ASCII value	Character	ASCII value	Character	ASCII value	Character
188	ƒ	211	ƒ	234	Ω
189	ƒ	212	ƒ	235	δ
190	ƒ	213	ƒ	236	∞
191	ƒ	214	ƒ	237	∅
192	ƒ	215	ƒ	238	€
193	ƒ	216	ƒ	239	∩
194	ƒ	217	ƒ	240	≡
195	ƒ	218	ƒ	241	±
196	—	219	■	242	≥
197	+	220	■	243	≤
198	ƒ	221	■	244	ƒ
199	ƒ	222	■	245	J
200	ƒ	223	■	246	÷
201	ƒ	224	α	247	≈
202	ƒ	225	β	248	°
203	ƒ	226	Γ	249	•
204	ƒ	227	π	250	•
205	≡	228	Σ	251	√
206	ƒ	229	σ	252	n
207	±	230	μ	253	²
208	ƒ	231	τ	254	■
209	ƒ	232	ϕ	255	(blank 'FF')
210	ƒ	233	⊖		

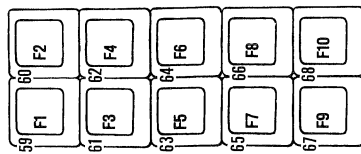
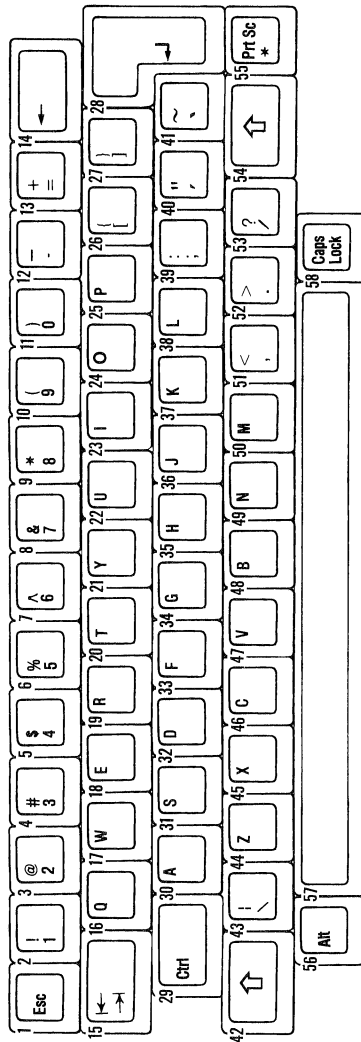
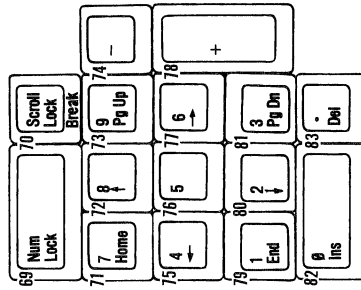
Extended Codes

When the INKEY\$ function is used on certain keys or key combinations that cannot be represented in the standard ASCII code, a two character extended code is returned. The first character contains a null character (ASCII code 0). The second character is usually, but not always, the scan code of the key that was pressed. Table B-2 lists the key or key sequence represented by the second character of the extended code.

Table B-2
Extended ASCII Codes

Second Character	Key / Key Sequence
3	NUL (null character
15	<-- (<Shift>/<Tab>)
16-25	<Alt>/Q, W, E, R, T, Y, U, I, O, P
30-38	<Alt>/A, S, D, F, G, H, J, K, L
44-50	<Alt>/Z, X, C, V, B, N, M
59-68	F1-F10 (when disabled as soft keys)
71	<Home>
72	Cursor Up
73	<Pg Up>
75	Cursor Left
77	Cursor Right
79	<End>
80	Cursor Down
81	<Pg Dn>
82	<Ins>
83	
84-93	F11-F20 (<Shift>/F1-F10)
94-103	F21-F30 (<Ctrl>/F1-F10)
104-113	F31-F40 (<Alt>/F1-F10)
114	<Ctrl>/<PrtSc>
115	<Ctrl>/<Cursor Left> (Previous Word)
116	<Ctrl>/<Cursor Right> (Next Word)
117	<Ctrl>/<End>
118	<Ctrl>/<Pg Dn>
119	<Ctrl>/<Home>
120-131	<Alt>/1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, =,
132	<Ctrl>/<Pg Up>

APPENDIX C KEYBOARD SCAN CODES



APPENDIX D GWBASIC RESERVED WORDS

The following is a list of reserved words used in GWBASIC. These words or abbreviations cannot be used as variable names.

ABS	DEFSNG	KEY	OUT	SQR
AND	DEFSTR	KILL	PAINT	STICK
ASC	DELETE	LEFT\$	PEEK	STOP
ATN	DIM	LEN	PEN	STR\$
AUTO	DRAW	LET	PLAY	STRIG
BEEP	EDIT	LINE	PMAP	STRING\$
BLOAD	ELSE	LIST	POINT	SWAP
BSAVE	END	LLIST	POKE	SYSTEM
CALL	EOF	LOAD	POS	TAB
CDBL	ERASE	LOC	PRESET	TAN
CHAIN	ERL	LOCATE	PRINT	THEN
CHDIR	ERR	LOF	PRINT#	TIME\$
CHR\$	ERROR	LOG	PSET	TIMER
CINT	EXP	LPOS	PUT	TO
CIRCLE	FIELD	LPRINT	RANDOMIZE	TROFF
CLEAR	FILES	LSET	READ	TRON
CLOSE	FIX	MERGE	REM	USING
CLS	FNxxxxxx	MID\$	RENUM	USR
COLOR	FOR	MKD\$	RESET	VAL
COM	FRE	MKDIR	RESTORE	VARPTR
COMMON	GET	MKI\$	RESUME	VARPTR\$
CONT	GOSUB	MKS\$	RIGHT\$	VIEW
COS	GOTO	MOD	RMDIR	WAIT
CSNG	HEX\$	MOTOR	RND	WEND
CSRLIN	IF	NAME	RSET	WHILE
CVD	IMP	NEW	RUN	WIDTH
CVI	INKEY\$	NEXT	SAVE	WINDOW
CVS	INP	NOT	SCREEN	WRITE
DATA	INPUT	OCT\$	SGN	WRITE#
DATE\$	INPUT#	ON	SIN	XOR
DEF	INPUT\$	OPEN	SOUND	
DEFDBL	INSTR	OPTION	SPACE	
DEFINT	INT	OR	SPC	

APPENDIX E MATHEMATICAL FUNCTIONS

Derived Functions

Functions that are not intrinsic to GWBASIC may be calculated as indicated in the following table.

Table E-1
Mathematical Functions

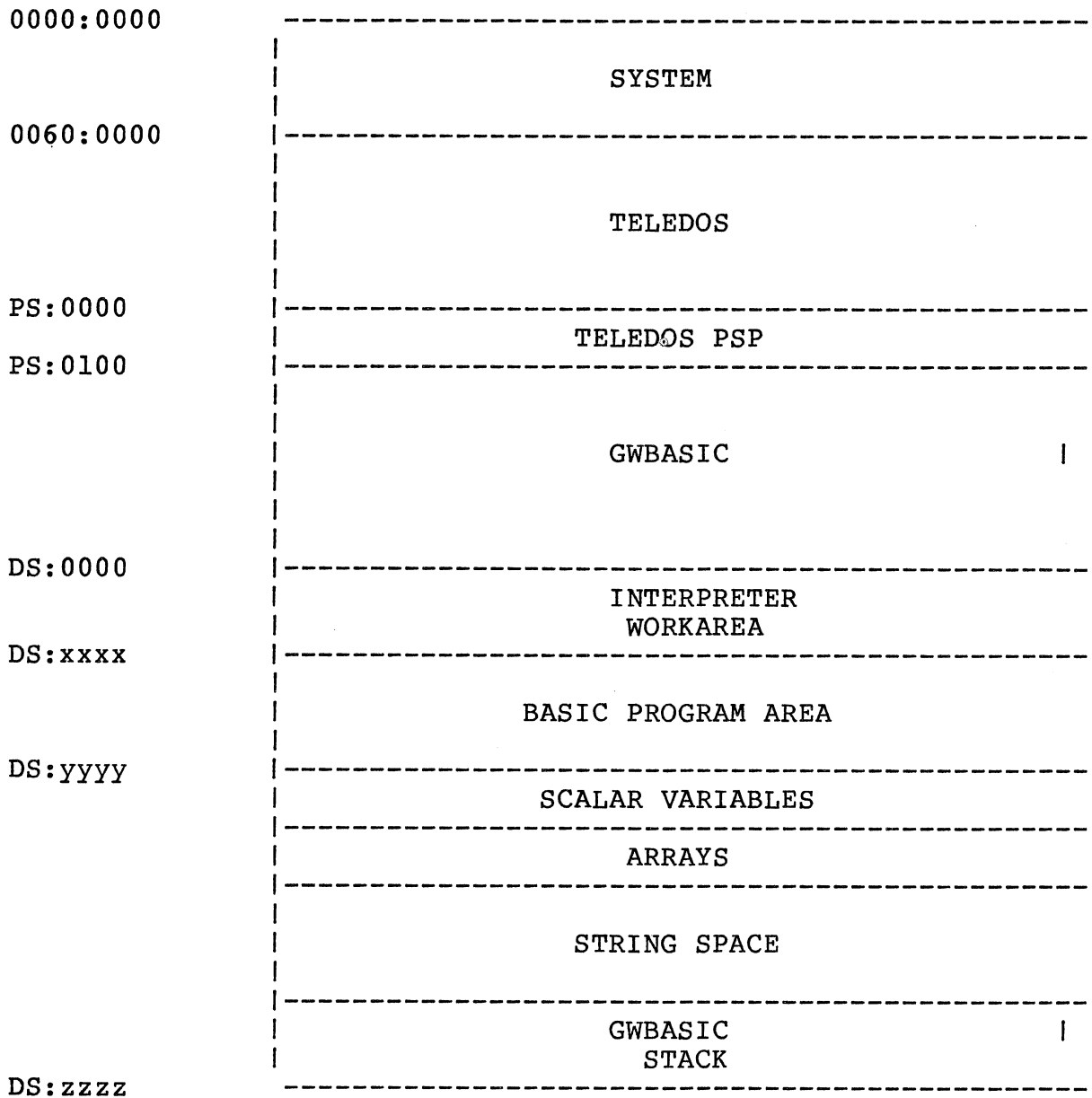
Function	GWBASIC Equivalent
SECANT	$SEC(X) = 1 / \cos(X)$
COSECANT	$CSC(X) = 1 / \sin(X)$
COTANGENT	$COT(X) = 1 / \tan(X)$
INVERSE SINE	$ARCSIN(X) = ATN(X / \sqrt{1 - X^2})$
INVERSE COSINE	$ARCCOS(X) = 1.5708 - ATN(X / \sqrt{1 - X^2})$
INVERSE SECANT	$ARCSEC(X) = ATN(X / \sqrt{X^2 - 1})$ $+ SGN(SGN(X) - 1) * 1.5708$
INVERSE COSECANT	$ARCCSC(X) = ATN(X / \sqrt{X^2 - 1})$ $+ (SGN(X) - 1) * 1.5708$
INVERSE COTANGENT	$ARCCOT(X) = 1.5708 - ATN(X)$
HYPERBOLIC SINE	$SINH(X) = (EXP(X) - EXP(-X)) / 2$
HYPERBOLIC COSINE	$COSH(X) = (EXP(X) + EXP(-X)) / 2$
HYPERBOLIC TANGENT	$TANH(X) = (EXP(X) - EXP(-X)) /$ $(EXP(X) + EXP(-X))$
HYPERBOLIC SECANT	$SECH(X) = 2 / (EXP(X) + EXP(-X))$
HYPERBOLIC COSECANT	$CSCH(X) = 2 / (EXP(X) - EXP(-X))$
HYPERBOLIC COTANGENT	$COTH(X) = (EXP(X) + EXP(-X)) /$ $(EXP(X) - EXP(-X))$
INVERSE HYPERBOLIC SINE	$ARCSINH(X) = LOG(X + \sqrt{X^2 + 1})$
INVERSE HYPERBOLIC COSINE	$ARCCOSH(X) = LOG(X + \sqrt{X^2 - 1})$
INVERSE HYPERBOLIC TANGENT	$ARCTANH(X) = LOG((1 + X) / (1 - X)) / 2$
INVERSE HYPERBOLIC SECANT	$ARCSECH(X) = LOG((\sqrt{1 - X^2} + 1) / X)$
INVERSE HYPERBOLIC COSECANT	$ARCCSCH(X) = LOG((SGN(X) * \sqrt{X^2 + 1} + 1) / X)$
INVERSE HYPERBOLIC COTANGENT	$ARCCOTH(X) = LOG((X + 1) / (X - 1)) / 2$

APPENDIX F TECHNICAL INFORMATION

This appendix contains technical information pertaining to GWBasic, including a memory map, a description of how GWBasic stores data, and a description of the GWBasic file control block.

MEMORY MAP

The following diagram is a memory map for GWBasic. Addresses are listed in hexadecimal in the format segment:offset.



- * PSP stands for the TeleDOS Program Segment Prefix
- * DS refers to GWBasic's Data Segment and is stored at location 0:510 and 0:511
- * Offset xxxx is stored in DS:30 and DS:31 (low byte, high byte)
- * Offset yyyy is stored in DS:358 and DS:359 (low byte, high byte)
- * Offset zzzz is at FFFF, or limited by the top of RAM memory

HOW VARIABLES ARE STORED

The following information describes how GWBasic stores scalar variables.

Type	Char1	Char2	Length	Characters	Data
0	1	2	3	4	4 + length

Type identifies the variable type

- 2 integer
- 3 string
- 4 single-precision
- 8 double-precision

Char1 is the first character of the variable name.

Char2 is the second character of the variable name.

Length is the number of additional characters in the variable name.

Characters are the remaining characters in the variable name.

Note, this means that at least three bytes are needed to store the variable name, and a maximum of 41 bytes can be used (for a 40 character name).

Data is the data area for the variable and can be 2, 3, 4, or 8 bytes long. This is the address value returned by the VARPTR function.

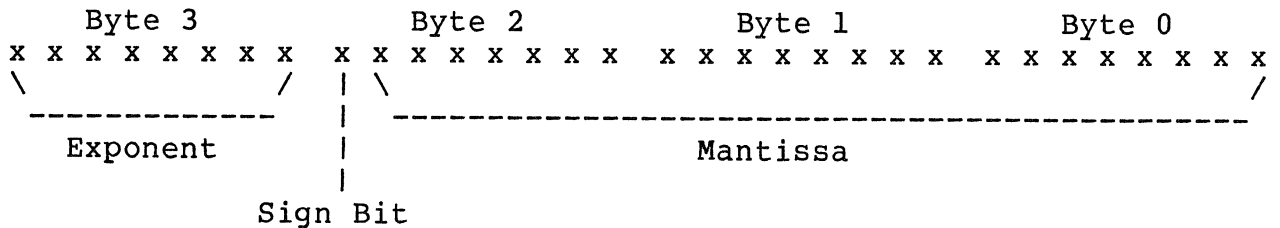
If the variable is numeric, the data area contains the actual value stored for the variable as follows:

- * Integer values are stored as hex values in two bytes, with low byte first, high byte second.
- * Single-precision values are stored in four bytes using a floating point binary format.
- * Double-precision values are stored in eight bytes using a floating point binary format.

If the variable is a string, the data area contains a three-byte string descriptor. The first byte contains the length of the string (0 to 255). The next two bytes contain the offset into the GWBasic data segment (DS) where the string value is stored. The offset is stored low byte first, high byte second.

Floating Point Binary Format

Single-precision numbers are store in the following format



The value is stored in a floating point binary format. This means the mantissa and exponent are stored as binary numbers. In a floating point binary format the mantissa is always greater than or equal to one (1) and less than two; therefore, the mantissa would always look like:

1.xxxxxxxxxxxxxxxxxxxxxxx

To save space, GWBasic assumes the 1, and only stores the fractional part of the mantissa; therefore, the 23 bits defined as the Mantissa above are the fractional part of the mantissa.

The high order bit of the third byte (Byte 2), is a sign bit for the entire number. A 0 indicates a positive number, a 1 indicates a negative number.

The Exponent is also stored in a binary format, but to allow easy comparison between numbers, the Exponent is biased with hex 81. This means that the value stored is the exponent plus hex 81.

Double-precision values are stored using the same format, but with 32 additional bits to store the mantissa. Byte 7 stores the biased exponent, the high order bit of byte 6 contains the sign bit, and byte 0 to byte 5 and the low order 7 bits of byte 6 store the mantissa. Byte 0 contains the low order bits of the mantissa while Byte 6 contains the seven high order bits.

GWBASIC File Control Block

GWBASIC constructs a file control block (FCB) for each open file. When you use the VARPTR function with a file number argument, the address returned is the offset into the GWBASIC data segment where this FCB is located. Table F-1 lists the information stored in the GWBASIC FCB.

**Table F-1
GWBASIC FCB**

Offset	Length	Description
0	1	The mode the file is open under: 1 Input only 2 Output only 4 Random access
1	38	TeleDOS file control block
39	2	The number of sectors read or written for sequential files. One plus the last record number read or written for random access files.
41	1	The number of bytes in the sector when read or written.
42	1	The number of bytes left in the input buffer.
43	3	Not used
46	1	Device number 0 default drive 1 drive A 2 drive B FA COM1: FB LPT2: FC LPT1: FE SCRN: FF KYBD:
47	1	Device width
48	1	The position of PRINT# in the buffer.
49	1	Not used for data files

50	1	Output used during tab expansion.
51	128	The physical data buffer used to transfer data between TeleDOS and GWBasic. This is the offset used to examine data for sequential I/O.
179	2	The record size set by the LEN argument in the OPEN statement. The default value is 128.
181	2	The current physical record number.
183	2	The current logical record number.
185	1	Not used
186	2	The position for PRINT #, INPUT #, and WRITE # for diskette files.
188	n	The actual file buffer for random access files. n is equal to the record size. This is the offset used to examine data for random access files.



INDEX

ABS function 5.3
 Absolute graphics
 coordinates 4.4
 Absolute value 5.3
 Active page 5.191
 Alt (Alternate) key 2.9
 AND operator 3.14
 Append 4.13
 Arctangent function 5.5
 Arithmetic operators 3.10
 Array variables 3.7-3.8,
 5.140
 ASC function 5.4
 ASCII codes B.9
 ATN function 5.5
 AUTO command 5.6

 Background 4.2
 BACKSPACE editor key 2.15
 Backspace key 2.9
 BASIC
 definition 1.1
 statement 3.1
 BEEP statement 5.9
 BLOAD statement 5-10
 Boolean operators 3.13-3.15
 Break function 2.11
 BSAVE statement 5.11

 CALL statement 5.12
 Caps Lock key 2.9
 Cartesian
 coordinates 5.156, 5.228
 CDBL function 5.12
 CHAIN statement 5.13
 Character set 3.2
 CHDIR 5.15
 CHR\$ function 5.16
 CINT function 5.17
 CIRCLE statement 5.18
 CLEAR statement 5.21
 CLOSE statement 5.23
 CLS statement 5.24
 COLOR statement,
 graphics 5.27
 text 5.25
 COM(n) statement 5.28
 Command definition 5.1
 Command level 2.1, 2.6, 2.15

Command line options 2.1
 Comments 3.1, 5.179
 COMMON statement 5.29
 Communications channel 5.137
 Communications event
 trapping 5.120
 Concatenation 3.17
 Constants
 defined 3.3
 numeric 3.3-3.4
 string 3.3
 CONT command 5.30
 Continuation, line 2.15, 3.2
 COS function 5.31
 Cosine function 5.31
 CSNG function 5.32
 CSRLIN function 5.33
 Ctrl (Control) key 2.8
 Current directory 4.9, 5.15
 Current graphics position 4.5
 CURSOR DOWN editor key 2.13
 CURSOR HOME editor key 2.12
 CURSOR LEFT editor key 2.12
 Cursor position 2.11
 CURSOR RIGHT editor key 2.13
 CURSOR UP editor key 2.12
 CVD function 5.34
 CVI function 5.34
 CVS function 5.34

 DATA statement 5.35
 DATE\$ function 5.36
 DATE\$ statement 5.37
 DEF FN statement 5.38
 DEF SEG statement 5.40
 DEFtype statement 5.42
 DEF USR statement 5.43
 Default device 4.6
 DEFDBL statement 5.42
 DEFINT statement 5.42
 DEFSNG statement 5.42
 DEFSTR statement 5.42
 DELETE command 5.44
 DELETE editor key 2.14
 Device names 4.6, Table 4-1
 DIM statement 5.45
 Dimensioning arrays 5.45
 Direct mode 2.6-2.7
 Display page 4.3

Division 3.10
 Division, integer 3.11
 Double-precision numbers 3.4-3.6, 5.12
 DRAW statement 5.47

 Echo function 2.11
 EDIT command 5.51
 Editor keys Table 2-3
 Editing programs 2.18-2.20
 Editor 2.11-2.18
 END statement 5.52
 End-of-file character 2.5, 4.12
 Enter key 2.9
 EOF function 5.53
 EQV operator 3.14
 ERASE statement 5.54
 ERL variable 5.55
 ERR variable 5.55
 Error codes 5.55, A.1
 Error messages A.1
 ERROR statement 5.57
 Error trapping 5.122
 ESCAPE editor key 2.14, 2.18
 Evaluation of operators
 arithmetic 3.10, 3.16
 logical 3.13-3.14, 3.16
 Event trapping
 communications 5.120
 error 5.122
 key 5.125
 light pen 5.127
 music 5.129
 joystick 5.131
 timer 5.133
 EXP function 5.59
 Exponential function 5.59
 Exponentiation 3.10
 Expressions 3.10
 Extended ASCII character codes B.12

 FIELD statement 4.14, 5.60
 File Control Block (FCB) 2.2, F.20
 File numbers 4.10, 5.135
 Filenames 4.7
 Files
 naming 4.6
 protected 4.10
 random access 4.13-4.17
 sequential 4.11-4.13
 FILES statement 5.62
 Filespec 2.2, 4.6, 5.2

FIX function 5.63
 Fixed-point numbers 3.3
 Floating point binary format F.19
 Floating-point numbers 3.4
 FOR...NEXT statement 5.64
 Foreground 4.2
 Formatting display output 5.164
 FRE function 5.67
 Function of special keys 3.2-3.3
 Function, definition of 5.1
 Function keys 2.10
 Functional operators 3.16
 Functions 3.16, 5.42

 GET statement, files 5.68
 GET statement, graphics 5.69
 GOSUB statement 5.72
 GOTO statement 5.74
 Graphics coordinates 4.4
 Graphics screen modes 4.3-4.6

 HEX\$ function 5.75
 Hexadecimal 2.4, 3.4, 5.75
 High resolution 4.4
 Home key 2.12

 IF...GOTO statement 5.76
 IF...THEN statement 5.76
 IF...THEN...ELSE statement 5.76
 IMP operator 3.14
 Indirect mode 2.6-2.7
 INKEY\$ function 5.78
 INP function 5.79
 INPUT statement 5.80
 INPUT# statement 5.82
 INPUT\$ function 5.83
 INSERT editor key 2.13
 INSTR function 5.84
 INT function 5.85
 Integer 3.3
 Integer division 3.11
 Interrupt keys 2.10

 KEY statement
 key trapping 5.86
 soft keys 5.88
 KEY trapping 5.86, 5.125
 KEY(n) statement 5.90
 Keyboard scan codes C.14
 Keyboard usage 2.7-2.11
 KILL statement 5.91

KYBD: 2.5
 Last point referenced 4.5
 LEFT\$ function 5.92
 LEN function 5.93
 LET statement 5.94
 Line continuation 2.15, 3.2
 Line editing 2.16-2.20
 LINE FEED editor key 2.15, 3.2
 Line format 3.1-3.2
 LINE INPUT statement 5.97
 LINE INPUT# statement 5.98
 Line length 2.15, 2.18, 3.2
 Line number generation 2.19, 5.6
 Line numbers 3.1
 LINE statement 5.95
 Line wrapping 2.14
 LIST command 5.99
 LLIST command 5.101
 LOAD command 5.102
 LOC function 5.103
 LOCATE statement 5.104
 LOF function 5.106
 LOG function 5.107
 Logical line 2.15
 Logical operators 3.13-3.15
 LPOS function 5.108
 LPRINT statement 5.109
 LPRINT USING statement 5.109
 LSET statement 5.111

 Mathematical functions E.16
 Medium resolution 4.3
 Memory map F.17
 MERGE command 5.112
 MID\$ function 5.113
 MID\$ statement 5.114
 MKD\$ function 5.116
 MKDIR command 5.115
 MKI\$ function 5.116
 MKS\$ function 5.116
 MOD operator 3.10-3.11
 Modes of operation 2.6
 Modulus arithmetic 3.10, 3.11
 Multiple statements
 on a line 3.1
 Multiplication 3.10
 Music 5.151
 Music event trapping 5.129

 NAME statement 5.117
 Negation 3.10
 NEW command 5.118

 NEXT WORD editor key 2.13
 NOT operator 3.13
 Notation conventions 1.2
 Numeric constants 3.3-3.4
 Numeric keypad 2.10
 Numeric precision 3.4
 Numeric type conversion 3.8-3.9
 Numeric variables 3.6

 OCT\$ function 5.119
 Octal 2.4, 3.4, 5.119
 Ok prompt 2.1
 ON COM(n) statement 5.120
 ON ERROR statement 5.122
 ON GOSUB statement 5.124
 ON GOTO statement 5.124
 ON KEY(n) statement 5.125
 ON PEN statement 5.127
 ON STRIG(n) statement 5.131
 ON TIMER statement 5.133
 OPEN statement 5.135
 OPEN COM statement 5.137
 Operators
 Arithmetic 3.10
 Boolean 3.13-3.15
 definition 3.9
 functional 3.16
 logical 3.13-3.15
 relational 3.12-3.13
 string 3.16
 OPTION BASE statement 5.140
 OR operator 3.14
 Order of evaluation
 arithmetic operators 3.10, 3.16
 logical operators 3.13, 3.16
 OUT statement 5.141
 Overlays 5.14

 PAINT statement 5.142
 Palette 5.27
 Path 4.8
 Pause function 2.11
 PEEK function 5.146
 PEN function 5.147
 PEN statement 5.149
 PEN event trapping 5.128
 Peripherals support 4.17
 Physical coordinates 5.156, 5.157
 Pixel 4.3
 PLAY function 5.150
 PLAY statement 5.151

PLAY ON statement 5.155
 PMAP function 5.156
 POINT function 5.157
 POINT statement 5.158
 POKE statement 5.159
 POS function 5.160
 Precedence
 arithmetic operators 3.10,
 3.16
 logical operators 3.13,
 3.16
 PRESET statement 5.161
 PREVIOUS word editor key 2.12
 PRINT statement 5.162
 PRINT USING statement 5.164
 PRINT# statement 5.169
 PRINT# USING statement 5.169
 Printer echo function 2.11
 Program editor 2.11-2.18
 Program line numbers 3.1
 Program lines 2.18, 3.1
 Prompt 2.1
 Protected files 4.10
 PrtSc (Print Screen) key 2.9
 PSET statement 5.172
 PUT statement, files 5.173
 PUT statement, graphics 5.174

 Random access files 4.13-4.17
 Random numbers 5.177, 5.187
 RANDOMIZE statement 5.177
 READ statement 5.178
 Records 2.3, 4.13, 5.136
 Relational operators
 3.12-3.13
 Relative graphics
 coordinates 4.5
 REM statement 5.179
 RENUM command 5.180
 Reserved words D.15
 RESET command 5.181
 Reset function 2.11
 RESTORE statement 5.182
 RESUME statement 5.183
 RETURN statement 5.184
 RIGHT\$ function 5.185
 RMDIR command 5.186
 RND function 5.187
 Root directory 4.9
 RSET statement 5.111
 RUN command 5.188

 SAVE command 5.189
 Scan codes, keyboard C.14
 SCREEN function 5.190

Screen line editor 2.11
 SCREEN statement 5.191
 SCRN: 2.6
 Segment address 5.40
 Sequential files 4.11-4.13
 SGN function 5.193
 Shift keys 2.9
 SIN function 5.194
 Sine function 5.194
 Single-precision numbers
 3.4-3.6, 5.32
 Soft keys 2.10, 5.88
 SOUND statement 5.195
 Space requirements for
 variables F.18
 SPACE\$ function 5.197
 SPC function 5.198
 Special character keys 3.2,
 Table 3-1
 Special I/O features 4.17,
 Table 4-3
 Special key combinations 2.11
 SQR function 5.199
 Square root function 5.199
 Standard input 2.2, 2.5
 Standard output 2.2, 2.5
 Starting GWBasic 2.1
 Statement, definition of 5.1
 STICK(n) function 5.200
 STOP statement 5.201
 STR\$ function 5.202
 STRIG function/statement 5.203
 STRIG(n) statement 5.205
 STRIG event trapping 5.131
 String constants 3.3
 String functions 3.17
 String operators 3.16
 String space F.17
 String variables 3.6
 STRING\$ function 5.206
 Subscripts 5.45, 5.140
 SWAP statement 5.207
 Syntax errors 2.21
 Syntax notation 1.2
 SYSTEM command 5.208

 TAB editor key 2.14
 TAB function 5.209
 TAN function 5.210
 Tangent function 5.210
 Text screen modes 4.1-4.3
 TIME\$ function 5.211
 TIME\$ statement 5.212
 Transcendental functions 2.3

Tree-structured
 directories 4.8
 Trace feature 5.214
 TROFF command 5.214
 TRON command 5.214
 Two's complement form 3.15
 Type declaration
 characters 3.5-3.6

 USR function 5.215


 VAL function 5.216
 Variables
 array 3.7-3.8
 names 3.5-3.7
 numeric 3.6
 passing with COMMON 5.29
 string 3.6
 VARPTR function 5.217
 VARPTR\$ function 5.219

 VIEW statement 5.220
 VIEW PRINT statement 5.223
 Viewport 5.220
 Visual page 5.191

 WAIT statement 5.224
 WEND statement 5.225
 WHILE statement 5.225
 WIDTH statement 5.226
 WINDOW statement 5.228
 WRITE statement 5.231
 WRITE# statement 5.232
 Writing programs 2.18-2.20

 XOR operator 3.14



 **TeleVideo Systems, Inc.**

125681-00 Rev. A1 B.R.

1170 Morse Avenue P.O. Box 3568 Sunnyvale, CA 94088

© TeleVideo 1/84. Printed in U.S.A.