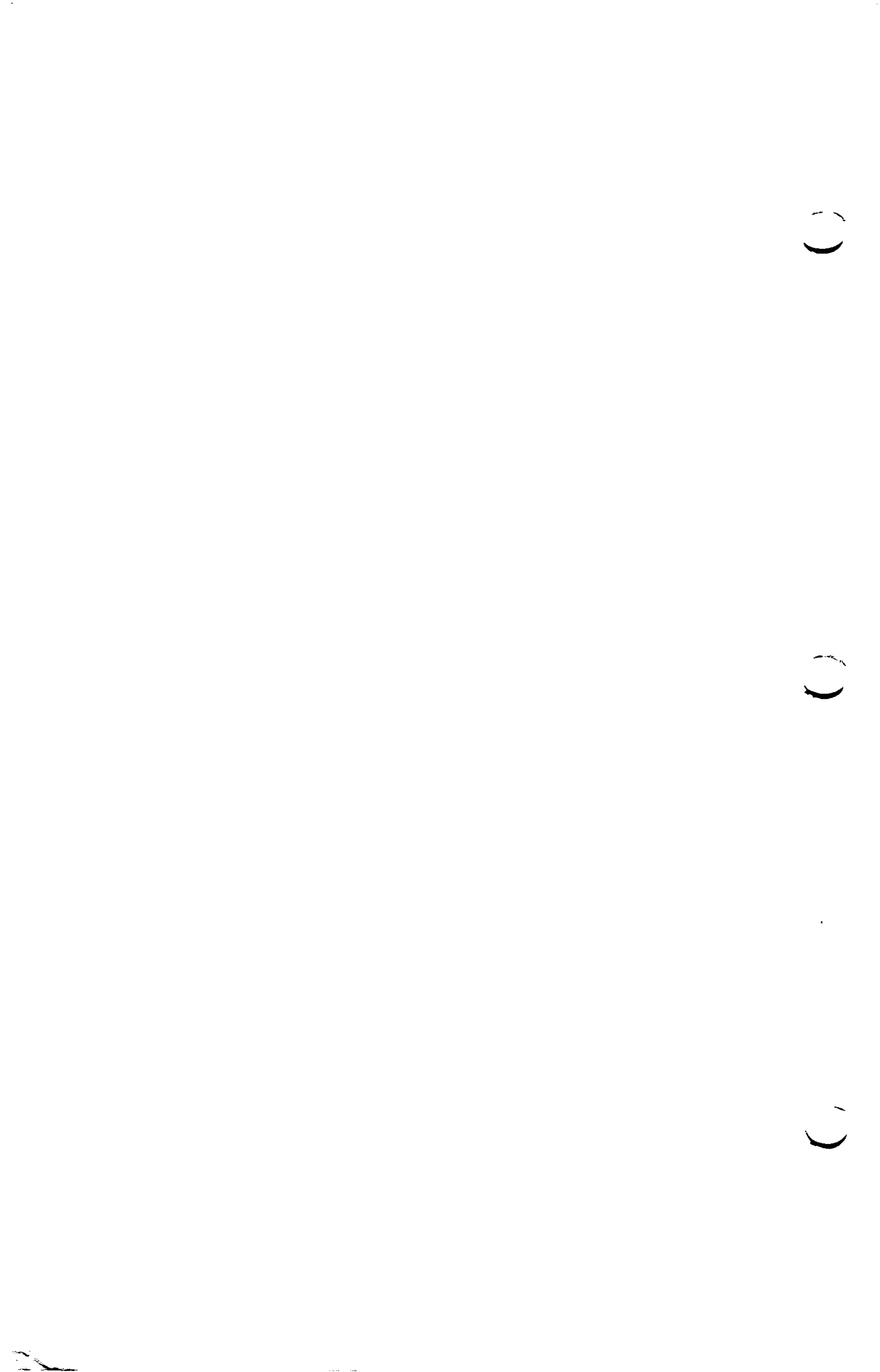
GW™-BASIC (Interpreter/Compiler) Reference Manual and User's Guide

For MS™-DOS

BI19317 May 1984



NCR Corporation is pleased to provide GW-BASIC software for implementation on your NCR Decision Mate V. Your GW-BASIC package contains an *NCR GW-BASIC Reference Manual*, a *GW-BASIC User's Guide* for either the GW-BASIC Interpreter or the GW-BASIC Compiler, and a disk which holds the following files:

NCR GW-BASIC (Interpreter)

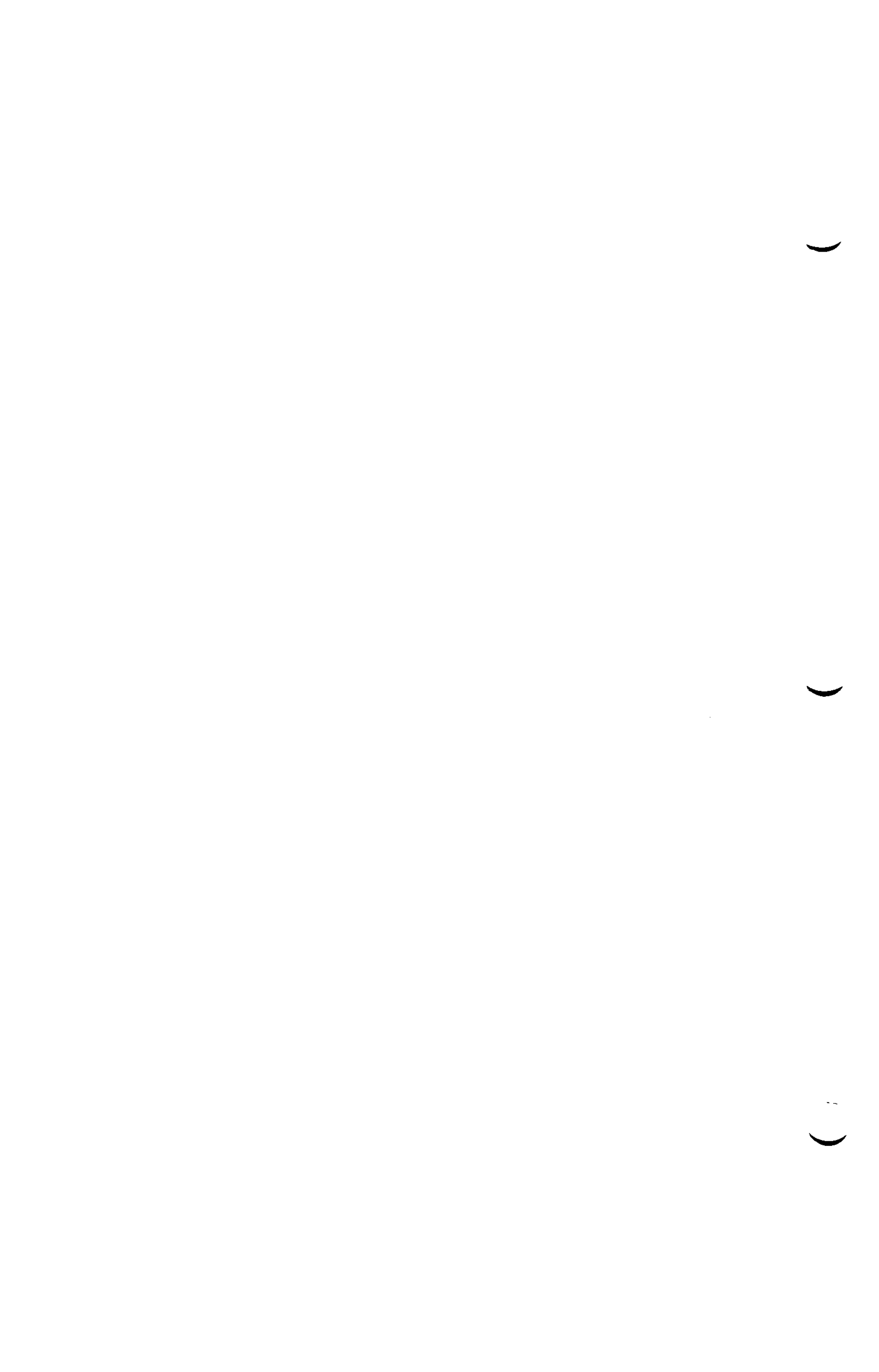
For MS™-DOS
Disk 1 of 1

GWBASIC.EXE
GWCONF.COM
DUMPCL.OBJ

NCR GW-BASIC Compiler

For MS™-DOS
Disk 1 of 1

GWBCOM.COM
BASCOMG.LIB
BASRUNG.LIB
BASRUNG.EXE
GWCONF.COM
DEMO.BAS
LINK.EXE



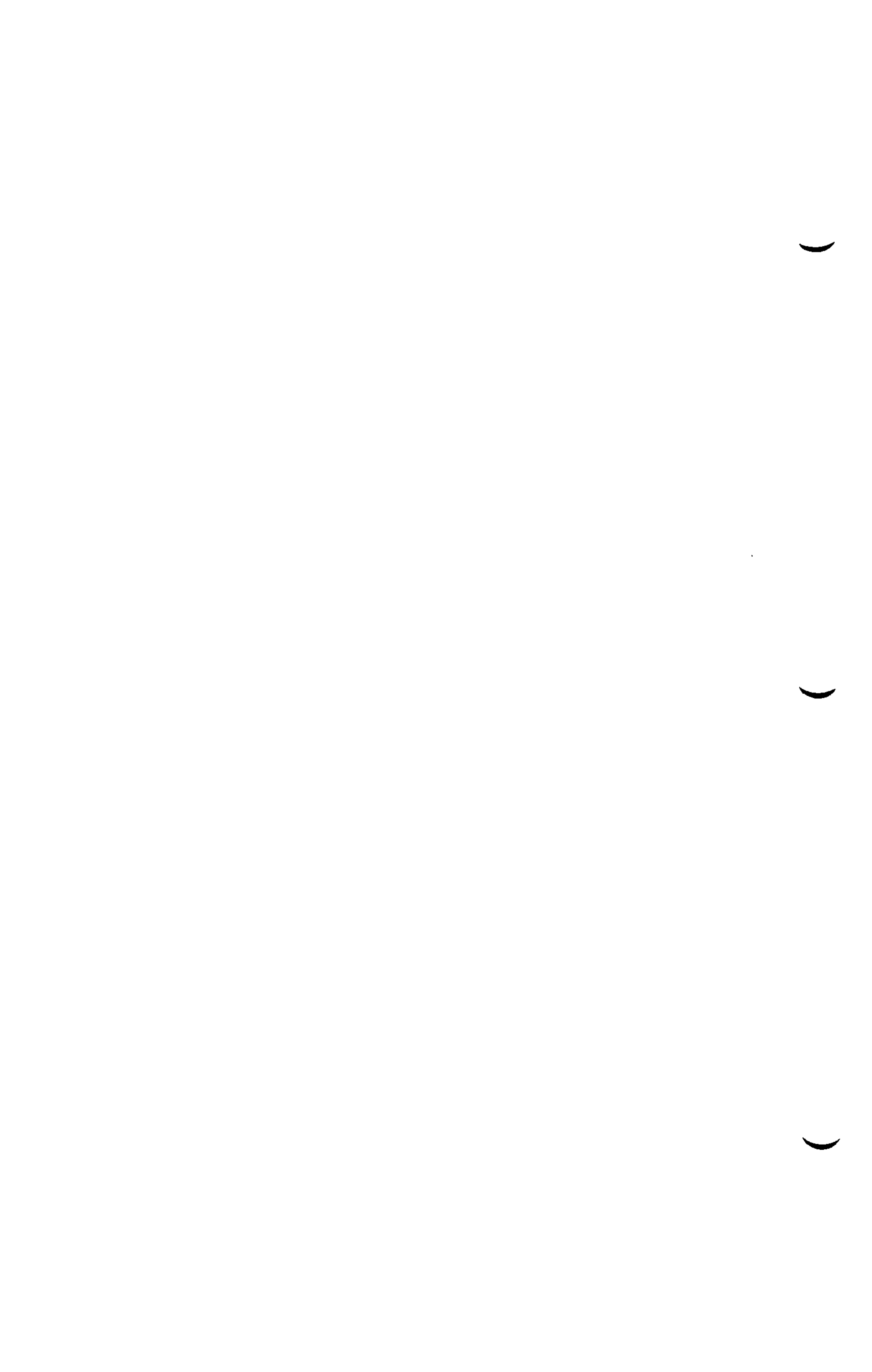
NCR GW-BASIC Compiler

The GW-BASIC Compiler program has been pre-installed for your NCR Decision Mate V.

No programmable function key assignments have been made. To define your own, see the KEY Statement in Chapter 4, Section 4.61, of your *NCR GW-BASIC Reference Manual.*

F1	
F2	
F3	
F4	
F5	
F6	
F7	
F8	
F9	
F10	
F11	
F12	
F13	
F14	
F15	

F16	
F17	
F18	
F19	
F20	

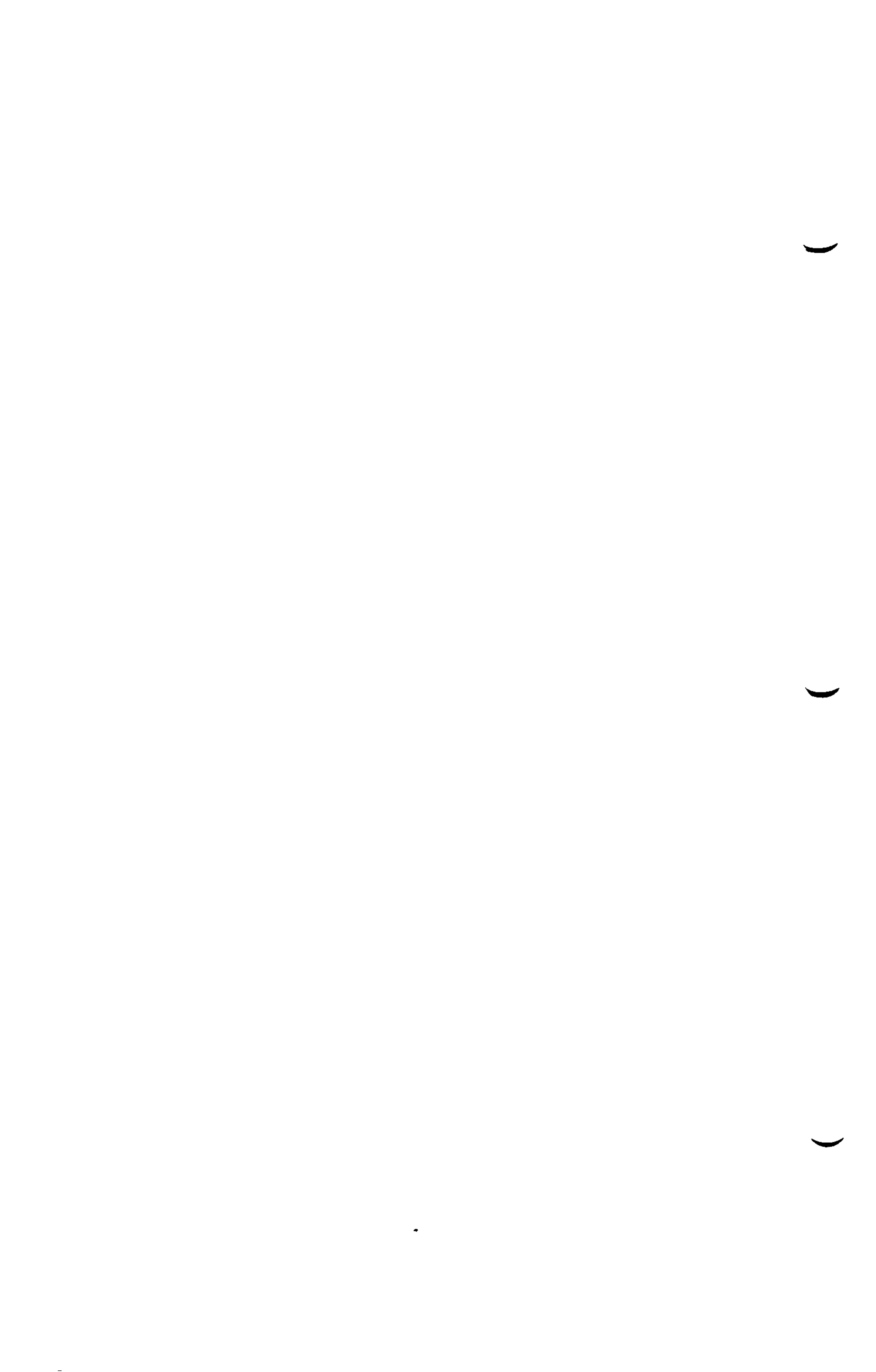


NCR GW-BASIC (Interpreter)

The GW-BASIC program has been pre-installed for your NCR Decision Mate V. The programmable function keys have been assigned the values which appear below. See the KEY Statement in Chapter 4, Section 4.61 of your *GW-BASIC REFERENCE MANUAL* for detailed instructions in utilizing these function keys.

F1	LOAD
F2	RUN
F3	CONT
F4	SAVE
F5	LIST
F6	EDIT
F7	TRON
F8	TROFF
F9	PRINT
F10	PRINT USING
F11	GOTO
F12	GOSUB
F13	IF
F14	THEN
F15	ELSE

F16	CHR$
F17	STRING$
F18	LINE
F19	CIRCLE
F20	DRAW



GW™-BASIC Reference Manual

For MS™-DOS

COPYRIGHT NOTICE

Copyright© 1983 by Microsoft Corporation, all rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a 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 the prior written permission of Microsoft Corporation.

TRADEMARKS

Microsoft and the Microsoft logo are registered trademarks of Microsoft Corporation. MS, GW, Music Macro Language, and Graphics Macro Language are trademarks of Microsoft Corporation. Teletype is a registered trademark of Teletype Corporation.

DISCLAIMER OF WARRANTY

NCR Corporation and Microsoft Corporation make no representations or warranties with respect to the contents hereof and specifically disclaim any implied warranties of merchantability or fitness for any particular purpose. Further, NCR Corporation and Microsoft Corporation reserve the right to revise this publication and to make changes from time to time in the content hereof without obligation to notify any person or organization of such revisions or changes.

The GW-BASIC Compiler Software and Manual are sold AS IS and without warranty as to performance. While NCR Corporation and Microsoft Corporation firmly believe this to be a high quality product, the user must assume all risks of using the program.

INTRODUCTION

NCR GW™-BASIC extends the capabilities of MS™-BASIC for MS™-DOS by providing graphics, sound and music, communications, device- independent input/output, event trapping and other enhancements for implementation on your NCR Decision Mate V.

This Reference Manual describes the capabilities that are provided with the NCR GW-BASIC Interpreter and the NCR GW-BASIC Compiler.

This Introduction explains how the manual is organized and gives the syntax notation used throughout the document.

Chapter 1, "GW-BASIC Features," briefly describes some of the special features that are supported by GW-BASIC.

Chapter 2, "GW-BASIC Editor," explains how programs are edited with GW-BASIC.

Chapter 3, "General Information About GW-BASIC," covers a variety of topics you need to know about when using GW-BASIC. Described here are GW-BASIC line format, character set, operators, etc. Some of these items differ from the interpreter to the compiler. Any differences will be pointed-out in this chapter.

Chapter 4, "GW-BASIC Commands, Statements, and Functions", provides detailed descriptions of the GW-BASIC language. Differences between the interpreted and compiled versions are noted.

Appendix A identifies error codes and messages and specifies those which are unique to the GW-BASIC Compiler.

Other Appendices list mathematical functions, ASCII character codes, and reserved words.

SYNTAX NOTATION

When commands are discussed in this document, the following notation will be followed:

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user-entered data. When the angle brackets enclose lowercase text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose uppercase text, the user must press the key named by the text; for example, <RETURN>.
{ }	Braces indicate that the user has 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.
¦	Vertical bars separate choices within braces. At least one of the entries separated by bars must be chosen unless the entries are also enclosed in square brackets.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

GW-BASIC Reference Manual

Contents

Introduction
Syntax Notation ii

Chapter 1 NCR GW-BASIC Features
1.1 Graphics 1-1
1.2 Screen Modes 1-1
1.2.1 Text Mode 1-1
1.2.2 Graphics Mode 1-2
1.2.3 X and Y Coordinates 1-3
1.3 Color Selection 1-4
1.4 Music Selection 1-6
1.5 Communications 1-7
1.6 Full Screen Editor 1-7
1.7 Peripheral Support 1-7
1.8 Event Trapping 1-7
1.9 Device-Independent Input/Output 1-8

Chapter 2 GW-BASIC Editor
2.1 Line Editing 2-1
2.2 EDIT Command 2-1
2.3 Full Screen Editor 2-2

Chapter 3 General Information About GW-BASIC
3.1 Modes of Operation 3-1
3.2 Line Format 3-1
3.3 Default Device 3-2
3.4 Active and Visual (Display) Pages 3-2
3.5 Character Set 3-2
3.5.1 Special Characters 3-2
3.5.2 Control Characters 3-3
3.6 Constants 3-4
3.6.1 String and Numeric Constants 3-5
3.6.2 Single/Double Precision Form for Numeric Constants 3-6
3.7 Variables 3-7
3.7.1 Variable Names and Declaration Characters . . 3-7
3.7.2 Array Variables 3-8
3.7.3 Space Requirements 3-9

3.8 Type Conversion 3-9
3.9 Expressions and Operators 3-11
3.9.1 Arithmetic Operators 3-11
3.9.1.1 Integer Division and Modulus Arithmetic . . 3-13
3.9.1.2 Overflow and Division By Zero 3-14
3.9.2 Relational Operators 3-15
3.9.3 Logical Operators 3-16
3.9.4 Functional Operators 3-18
3.9.5 String Operators 3-18
3.10 Error Messages 3-19

Chapter 4— GW-BASIC Commands, Statements, and Functions

4.1 ABS Function 4-4
4.2 ASC Function 4-5
4.3 ATN Function 4-6
4.4 AUTO Command 4-7
4.5 BEEP Statement 4-8
4.6 BLOAD Statement 4-9
4.7 BSAVE Statement 4-11
4.8 CALL Statement 4-13
4.9 CALLS Statement 4-15
4.10 CDBL Function 4-16
4.11 CHAIN Statement 4-17
4.12 CHR$ Function 4-21
4.13 CINT Function 4-22
4.14 CIRCLE Statement 4-23
4.15 CLEAR Statement 4-26
4.16 CLOSE Statement 4-28
4.17 CLS Statement 4-29
4.18 COLOR Statement 4-30
4.19 COM Statement 4-32
4.20 COMMON Statement 4-33
4.21 CONT Command 4-36
4.22 COS Function 4-37
4.23 CSNG Function 4-38
4.24 CSRLIN Function 4-39
4.25 CVI, CVS, CVD Functions 4-40
4.26 DATA Statement 4-41
4.27 DATE$ Statement 4-42
4.28 DATE$ Function 4-43
4.29 DEF FN Statement 4-44
4.30 DEFINT/SNG/DBL/STR Statements 4-46
4.31 DEF SEG Statement 4-48

4.32 DEF USR Statement 4-49
4.33 DELETE Command 4-50
4.34 DIM Statement 4-51
4.35 DRAW Statement 4-53
4.36 EDIT Command 4-56
4.37 END Statement 4-57
4.38 EOF Function 4-58
4.39 ERASE Statement 4-60
4.40 ERR and ERL Variables 4-61
4.41 ERROR Statement 4-62
4.42 EXP Function 4-64
4.43 FIELD Statement 4-65
4.44 FILES Statement 4-68
4.45 FIX Function 4-69
4.46 FOR...NEXT Statement 4-70
4.47 FRE Function 4-73
4.48 GET Statement 4-74
4.49 GET and PUT Statements 4-75
4.50 GOSUB...RETURN Statements 4-80
4.51 GOTO Statement 4-82
4.52 HEX$ Function 4-83
4.53 IF...THEN [...ELSE]/IF...GOTO Statements . . 4-84
4.54 INKEY$ Function 4-87
4.55 INP Function 4-88
4.56 INPUT Statement 4-89
4.57 INPUT# Statement 4-91
4.58 INPUT$ Function 4-92
4.59 INSTR Function 4-93
4.60 INT Function 4-94
4.61 KEY Statement 4-95
4.62 KEY(N) Statement 4-98
4.63 KILL Statement 4-100
4.64 LCOPY Statement 4-101
4.65 LEFT$ Function 4-105
4.66 LEN Function 4-106
4.67 LET Statement 4-107
4.68 LINE Statement 4-108
4.69 LINE INPUT Statement 4-111
4.70 LINE INPUT# Statement 4-112
4.71 LIST Statement 4-113
4.72 LOAD Command 4-115
4.73 LOC Function 4-116
4.74 LOCATE Statement 4-117

4.75 LOF Function 4-119
4.76 LOG Function 4-120
4.77 LPOS Function 4-121
4.78 LPRINT and LPRINT USING Statements . . . 4-122
4.79 LSET and RSET Statements 4-123
4.80 MERGE Command 4-124
4.81 MID$ Statement 4-125
4.82 MID$ Function 4-126
4.83 MKI$, MKS$, MKD$ Functions 4-127
4.84 NAME Statement 4-128
4.85 NEW Command 4-129
4.86 OCT$ Function 4-130
4.87 ON COM(n) Statement 4-131
4.88 ON ERROR GOTO Statement 4-133
4.89 ON...GOSUB and ON...GOTO Statements . . . 4-134
4.90 ON KEY(n) Statement 4-135
4.91 ON STRIG Statement 4-137
4.92 OPEN Statement 4-139
4.93 OPEN COM Statement 4-141
4.94 OPTION BASE Statement 4-145
4.95 OUT Statement 4-146
4.96 PAINT Statement 4-147
4.97 PEEK Function 4-149
4.98 PLAY Statement 4-150
4.99 POINT Function 4-153
4.100 POKE Statement 4-154
4.101 POS Function 4-155
4.102 PRESET Statement 4-156
4.103 PRINT Statement 4-157
4.104 PRINT USING Statement 4-160
4.105 PRINT# and PRINT# USING Statements . . . 4-166
4.106 PSET Statement 4-169
4.107 PUT Statement 4-170
4.108 RANDOMIZE Statement 4-171
4.109 READ Statement 4-172
4.110 REM Statement 4-174
4.111 RENUM Command 4-175
4.112 RESET Command 4-177
4.113 RESTORE Statement 4-178
4.114 RESUME Statement 4-179
4.115 RETURN Statement 4-181
4.116 RIGHT$ Function 4-182
4.117 RND Function 4-183

4.118 RUN Statement/Command 4-184
4.119 SAVE Command 4-186
4.120 SCREEN Function 4-187
4.121 SCREEN Statement 4-188
4.122 SGN Function 4-189
4.123 SIN Function 4-190
4.124 SOUND Statement 4-191
4.125 SPACE$ Function 4-194
4.126 SPC Function 4-195
4.127 SQR Function 4-196
4.128 STICK Function 4-197
4.129 STOP Statement 4-198
4.130 STR$ Function 4-199
4.131 STRIG Statement/Function 4-200
4.132 STRING$ Function 4-202
4.133 SWAP Statement 4-203
4.134 SYSTEM Command 4-204
4.135 TAB Function 4-205
4.136 TAN Function 4-206
4.137 TIME$ Statement 4-207
4.138 TIME$ Function 4-208
4.139 TRON/TROFF Statements/Commands 4-209
4.140 USR Function 4-210
4.141 VAL Function 4-212
4.142 VARPTR Function 4-213
4.143 VARPTR$ Function 4-214
4.144 WAIT Statement 4-215
4.145 WHILE...WEND Statement 4-216
4.146 WIDTH Statement 4-218
4.147 WRITE Statement 4-220
4.148 WRITE# Statement 4-221

Appendix A—Error Codes and Error Messages

A.1 Runtime Error Messages A-1
A.2 Compiler Invocation Error Messages A-11
A.3 Compiletime Error Messages A-13
A.3.1 Severe Errors A-13
A.3.2 Warning Errors A-17
A.4 MS-LINK Error Messages A-17

Appendix B—Mathematical Functions

Appendix C—ASCII Character Codes

Appendix D—GW-BASIC Reserved Words

NCR-GW-BASIC Features

This chapter describes the special features that are part of NCR GW-BASIC. These features include graphics, sound and music, peripherals support, device-independent I/O, event trapping, and others.

1.1 GRAPHICS

GW-BASIC allows you to use color, draw various figures, and perform animation. The statements and functions that are used for graphics include:

CIRCLE	PAINT
COLOR	POINT
DRAW	PRESET
GET and PUT	PSET
LINE	SCREEN

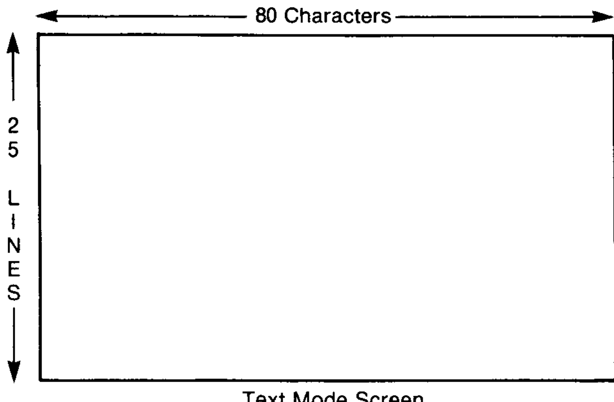
These statements and functions are described in Chapter 4.

1.2 SCREEN MODES

NCR GW-BASIC operates in either of two modes. Mode 0 is text mode; it is the default (usual) mode. Mode 1 is graphics mode. You must switch the system into this mode (with the SCREEN statement) whenever you use certain statements. Why you must do this becomes clear if you know more about how GW-BASIC Interpreter and Compiler handle screen input and output.

1.2.1 Text Mode

In text mode, the software considers the screen to have 25 lines (from top to bottom) and 80 characters per line. (Line 25 is reserved for programmable function key display.)

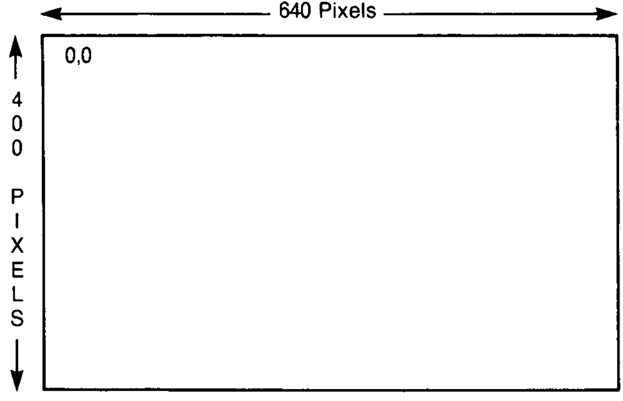


Text Mode Screen

When the software displays a character you enter on the keyboard, it internally translates the key you press and displays its image at the cursor position. In text mode, you are working with a specific character set: those characters you see on your keyboard.

1.2.2 Graphics Mode

Graphics mode is more sophisticated. To allow you to draw pictures and other shapes, the software considers the screen to be made up of *pixels.* A pixel is simply a dot on the screen. Your NCR DECISION MATE V has 640 pixels across and 400 pixels down.



Graphics Mode Screen

Of the statements and functions available in GW-BASIC, the following ones must be used in graphics mode. (Being "in graphics mode" simply means you have entered a screen statement specifying mode 1.)

CIRCLE	LINE	GET
DRAW	PRESET	PUT
PAINT	PSET	POINT (Function)

Remember you may use any other BASIC statements while in graphics mode, but you *must* be in graphics mode to use any graphics statement. Because screen handling for a graphics is more complex than for text, always shift back to text mode when the graphics portion of your program is complete and always be in text mode when editing your program.

1.2.3 X and Y Coordinates

The graphics statements require both an x and a y coordinate that describe where on the screen you want to begin to draw. The x coordinate is the horizontal position on the screen; the y coordinate is the vertical position. 0,0 is the first pixel position in the upper left-hand corner of the screen.

With most graphics statements, you can specify the coordinates in either of two forms: an absolute form where x,y specify the exact position, or an offset form where x,y are the offset values from the last point referenced. When specifying the coordinates in offset form, you must include the word STEP to let the software know you are "stepping" from a previously established point.

Consider the following two examples:

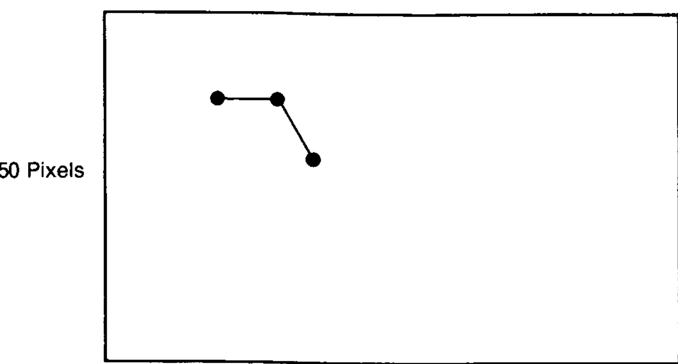
```
10 SCREEN 1
20 LINE (100,100)-(150,100)
30 LINE (150,100)-(200,150)
```

Example 1 Absolute Form

```
10 SCREEN 1
20 LINE (100,100)-STEP (50,0)
30 LINE-STEP (50,50)
```

Example 2 Offset Form

Both examples produce the following lines on your screen. Both specify the starting pixel location at 100,100.



1.3 COLOR SELECTION

If you have a color screen, you can select different colors for the foreground (the character or graphics image) and the background (the screen itself). On the monochrome model, the characters are green displayed on a black background. You specify the colors you want with the Color statement or, if drawing a graphics image, with the graphics statement.

The colors available on your NCR DECISION MATE V are shown in the following list. (The numbers are used to indicate the color on the graphics statements.)

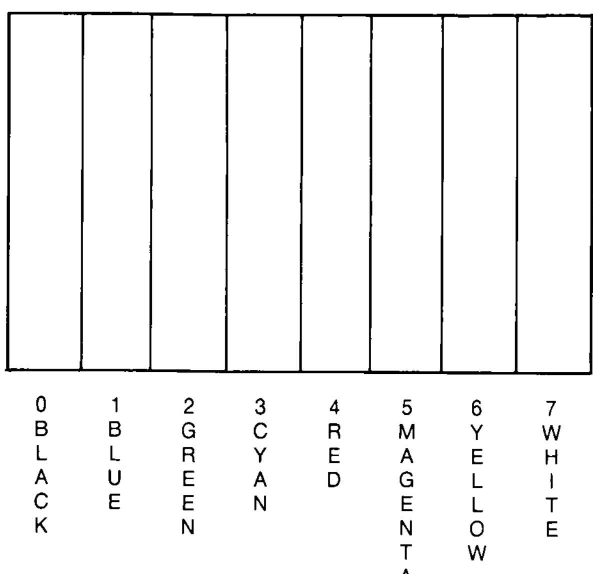
0 = black	4 = red
1 = blue	5 = magenta
2 = green	6 = yellow
3 = cyan	7 = white

When using colors, you should be aware of how they are stored in memory, especially if you are going to print out your screen image. This information may affect your decision on what colors you use for your images.

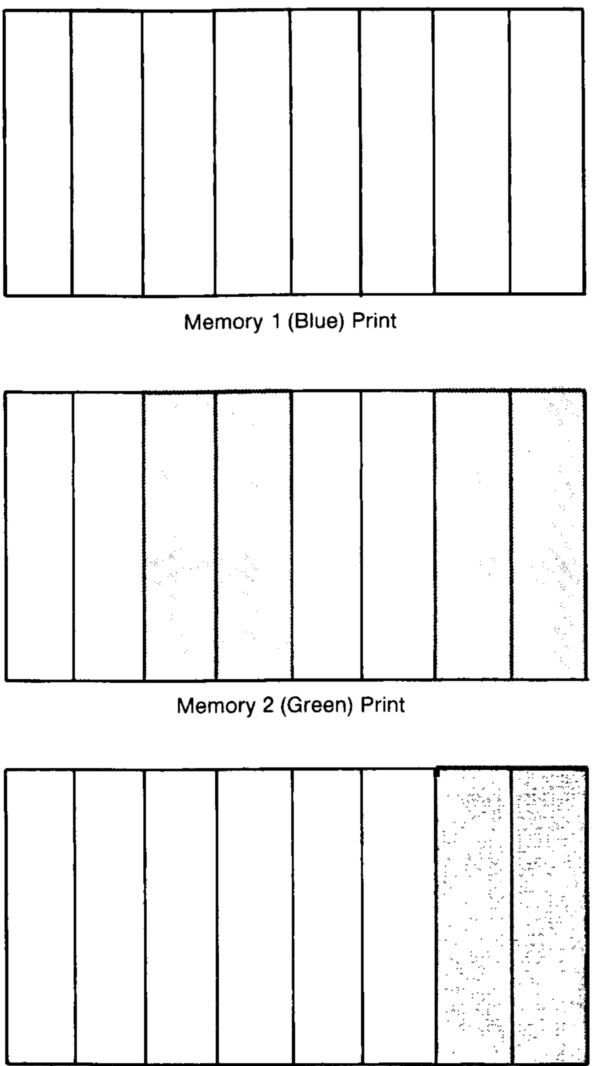
The various colors are stored in different memories and only one memory may be printed at a time; therefore, depending on which colors you use, you may or may not get a complete image printout. The following table lists in which memory a color is stored. Note that the colors are grouped by primary color and some colors are stored in more than one memory.

Memory 1 Blue	Memory 2 Green	Memory 4 Red
Cyan	Cyan	
Magenta		Magenta
	Yellow	Yellow
White	White	White

You print the screen image by specifying the memory you want printed either with the GW-BASIC Configure routine or by including special coding within your program. (Both methods are discussed later in your *GW-BASIC User's Guide.*) For now assume that the following image is on your screen. Each of the 8 colors is a vertical bar.



Now look at how the image would be printed, depending on the memory specified.



Memory 1 (Blue) Print

Memory 2 (Green) Print

Memory 4 (Red) Print

1.4 MUSIC SELECTION

NCR GW-BASIC includes statements that allow you to play music (or even just make noise). These statements are BEEP, SOUND, and PLAY. Of the statements, PLAY is the most powerful, since with it you can generate an entire musical piece with one statement. SOUND, on the other hand, generates a single note, while BEEP does exactly what its name suggests. The above statements are discussed in-detail in Chapter 4.

1.5 COMMUNICATIONS

Using NCR GW-BASIC, you can communicate with any other computer, printer, or device that uses an RS-232 asynchronous interface. To implement communications, you must first describe the communications device with the GW-BASIC Configure routine. See your NCR GW-BASIC or NCR GW-BASIC Compiler User's Guide for detailed information concerning communications implementation on your NCR Decision Mate V.

1.6 FULL SCREEN EDITOR

Like BASIC, GW-BASIC operates in either direct or indirect mode and uses the same programming conventions. As examples, the BASIC rules for data types, data entry, program lines also apply in GW-BASIC.

With GW-BASIC, however, you have a full screen editor. This feature simply means you can quickly edit any line of text *anywhere on your screen.*

Note that generally, you can enter and edit text only with the GW-BASIC Interpreter. With the GW-BASIC Compiler, however, you can use some of the line editing capabilities when you are entering text in response to an INPUT statement.

GW-BASIC editor features are more fully explained in Chapter 2 of this manual.

1.7 PERIPHERAL SUPPORT

The joystick feature is available as a peripheral device for implementation on the NCR Decision Mate V. The joystick feature is supported by the STICK function and the STRIG statement/function. (See Chapter 4 for complete descriptions of STICK and STRIG.)

1.8 EVENT TRAPPING

Event trapping allows a program to transfer control to a specific program line when a certain event occurs. Control is transferred as if a GOSUB statement had been executed to the trap routine starting at the specified line number. The trap routine, after servicing the event, executes a RETURN statement that causes the program to resume execution at the place where it was when the event trap occurred.

The events that can be trapped are receipt of characters from communications port (ON COM), function key activation (ON KEY), and joystick trigger activation (ON STRIG).

For more details on individual statements, see Chapter 4.

1.9 DEVICE-INDEPENDENT INPUT/OUTPUT

GW-BASIC provides device-independent input/output that works with various operating systems, stand-alone systems, disk-based RAM systems, non-disk ROM systems, and hooked systems. Any modifications that may be required are minimal.

The following statements, commands, and functions support device-independent I/O (see individual descriptions in Chapter 4):

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

GW-BASIC Editor

GW-BASIC provides three ways to enter and edit text: you can use the line editing capabilities, issue an EDIT command to place you in edit mode, or use the full screen editor. Generally, you can enter and edit text only with the GW-BASIC Interpreter. With the GW-BASIC Compiler, however, you can use some of the line editing capabilities when you are entering text in response to an INPUT statement. See Chapter 4 for information concerning the EDIT command and INPUT statement.

2.1 LINE EDITING

If the cursor is currently on a line, you can make the following changes. If you are entering a line in response to an INPUT statement, you can use the first two items in the list:

1. Delete an incorrect character from the line that is being typed, by pressing the backspace key or Control-H. Both these actions delete the last character entered, or the character to the left of the cursor.

2. Delete the entire line that is being typed by pressing Control-U.

3. Correct program lines for a program that is currently in memory by retyping the line, using the same line number. GW-BASIC will automatically replace the old line with the new one.

4. Delete the entire program currently residing in memory by entering the NEW command. NEW is usually used to clear memory prior to entering a new program. See Chapter 4 for more information about NEW.

2.2 EDIT COMMAND

The EDIT command places the cursor on a specified line so that changes can be made to the line. See Chapter 4 for a description of the EDIT command.

2.3 FULL SCREEN EDITOR

Like BASIC, GW-BASIC operates in either direct or indirect mode and uses the same programming conventions.

With GW-BASIC, however, you have a full screen editor. This feature simply means you can quickly edit any line of text *anywhere on your screen.*

Table 1 lists the keys that control the movement of the cursor. In some cases, you have a choice of keys; use the one most comfortable for your entry. When a combination of keys must be used (as with CONTROL-J), hold down the CONTROL key and press the second key.

Besides providing full screen movement, the editor also allows for more efficient editing. Use the LIST statement to modify existing program lines, being sure to RETURN to store the modified line in the program.

- Occasionally, GW-BASIC may return to direct mode with the cursor positioned on a line containing a message, such as OK. When this occurs, the line is automatically erased. If it were not erased and you entered RETURN, the message would be given to GW-BASIC for interpretation and a syntax error would result. BASIC messages end with hexadecimal FF to distinguish them from user text.
- After you alter a line, you do not need to move the cursor to the end of the logical line before typing RETURN. The editor remembers where each logical line ends and transfers the line, even if RETURN is typed at the beginning of a line.

The editor also functions during program execution. If a syntax error occurs, GW-BASIC automatically enters edit at the line that caused the error. For example,

```
10 A = 2$12
RUN
? Syntax Error in 10
10 A = 2$12
```

The editor displays the line in error and positions the cursor under the digit 1. You would move the cursor to the dollar sign ($) and change it to an up-arrow (↑), followed by a RETURN. The corrected line is now stored back in the program.

In this example, storing the line back in the program causes all variables to be lost. Had you wanted to examine the contents of some variable before making the change, Control-C would be typed to

return to Direct Mode. The variables would be preserved since no program line was changed, and after you were satisfied, you could edit the line and re-run the program.

Key(s) (hex/dec value)	Name	Description
↵ or CONTROL-M (0D/13)	RETURN	Sends the line (up to the cursor) to GW-BASIC for interpretation.
CONTROL-J (0A/10)	LINE FEED	Moves the cursor to the first position on the next line. (Scrolling will occur if cursor is on line 24.)
↖ or CONTROL-K (0B/11)	HOME	Moves the cursor to the upper left-hand corner of the screen.
↑ or CONTROL-^ (1E/30)	CURSOR UP	Moves the cursor up 1 line.
↓ or CONTROL-0 (1F/31)	CURSOR DOWN	Moves the cursor down 1 line.
← or CONTROL-] (1D/29)	CURSOR LEFT	Moves the cursor 1 position left. When advanced beyond the left of the screen, the cursor is moved to the right side of the preceding line.
→ or CONTROL-\ (1C/28)	CURSOR RIGHT	Moves the cursor 1 position right. When advanced beyond the right of the screen, the cursor is moved to the left side of the next line.
CONTROL-→ or CONTROL-F (06/06)	NEXT WORD	Moves the cursor right, to the next word. "Next word" is the next character to the right of the cursor (or on the next line) in the set A-Z or 0-9.
CONTROL-← or CONTROL-B (02/02)	PREVIOUS WORD	Moves the cursor left, to the previous word. "Previous word" is the next character to the left of the cursor (or on the previous line) in the set A-Z or 0-9.

Table 1 Editing Keys (1 of 3)

Key(s) (hex/dec value)	Name	Description
⇤ or CONTROL-H (08/08)	BACK SPACE	Deletes the last character entered, or the character to the left of the cursor. (If the character is in the first column, it moves off the screen.) All characters to the right of the cursor are moved left 1 position. If a logical line extends beyond a physical line, subsequent characters are shifted left and up to fill the line.
CONTROL-E (05/05)	ERASE TO END	Erases to the end of a logical line from the current cursor position.
CONTROL-↖ or CONTROL-L (0C/12)	CLEAR SCREEN	Clears the screen and positions the cursor in the upper left-hand corner of the screen.
CONTROL-U (15/21)	ESCAPE	Erases the entire logical line.
CONTROL-C (03/03)	BREAK	Returns to direct mode without saving any changes that were made to the current line.
CONTROL-T (14/20)	FUNCTION KEY DISPLAY	Advances the display of function keys on line 25.
CONTROL-Q (11/17)	MARK LINE	Marks a line for deletion.
CONTROL-R (12/18)	INSERT	Turns insert mode either on or off. (Insert mode is used to place characters between characters in a line.) If insert mode is off, pressing this key turns it on; if insert mode is on, pressing this key turns it off. When in insert mode, characters following the cursor are moved to the right as typed characters are inserted at the current position. For each keystroke, the cursor moves one position to the right. If characters (or blanks) move off the right side of the screen, they are inserted from the left on subsequent lines. When out of insert mode, characters typed replace existing characters on the line.

Table 1 Editing Keys (2 of 3)

Key(s) (hex/dec value)	Name	Description
CONTROL-I (09/09)	TAB	When out of insert mode, pressing the key moves the cursor over characters until the next tab stop is reached. Tab stops occur every 8 character positions. When in insert mode, pressing the key causes blanks to be inserted from the current cursor position to the next tab stop.
CONTROL-N (0E/14)	END	Moves the cursor to the end of the logical line. Characters typed from this position are appended to the line.

Table 1 Editing Keys (3 of 3)

General Information About GW-BASIC

For full instructions for initializing GW-BASIC or GW-BASIC Compiler on your NCR Decision Mate V, see you *NCR GW-BASIC (Interpreter)* or *NCR GW-BASIC Compiler User's Guide.*

3.1 MODES OF OPERATION

GW-BASIC Interpreter may be used in either of two modes: direct mode or indirect mode. *These modes do not apply to the GW-BASIC Compiler.*

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

Indirect mode is used for entering programs. Program lines are preceded by line numbers and may be stored in memory. The program stored in memory is executed by entering the RUN command.

3.2 LINE FORMAT

GW-BASIC program lines have the following format (square brackets indicate optional input):

nnnnn BASIC statement [:BASIC statement...] <carriage return>

More than one GW-BASIC statement may be placed on a line, but each must be separated from the last by a colon.

A GW-BASIC program line always begins with a line number and ends with a carriage return. 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. Line numbers must be in the range 0 to 65529.

With the interpreter, a line may contain a maximum of 255 characters. With the compiler, the maximum number of characters per line is 253.

With the interpreter, you can extend a logical line over more than one physical line by using the <linefeed> key. <linefeed> lets you continue typing a logical line on the next physical line without entering a <carriage return>.

With the compiler, the line continuation character is an underscore (_). Enter the underscore as the last character before you press <RETURN> to drop down to the next line. The underscore removes the significance of the carriage return in the <carriage return><linefeed> sequence that ends each line, so that just the linefeed is presented to the compiler.

A period (.) may be used in EDIT, LIST, AUTO, and DELETE commands to refer to the current line. *Note that these commands work only with the interpreter, not with the compiler.*

3.3 DEFAULT DEVICE

When a filespec is given (in commands or statements such as FILES, OPEN, KILL), the default disk drive is the one that was the default in MS-DOS before GW-BASIC was invoked.

3.4 ACTIVE AND VISUAL (DISPLAY) PAGES

Every command that reads to or writes from the screen is actually reading/writing from or to the active page. The visual, or display, page is the active page that is shown on the terminal screen.

The size of these pages is set by the SCREEN statement. (Section 4.121.)

3.5 CHARACTER SET

The GW-BASIC character set consists of alphabetic characters, numeric characters, and special characters.

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

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

3.5.1 Special Characters

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

Character	Action
	Blank
=	Equals sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
↑	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At sign
	Underscore
<backspace>	Deletes last character typed.
<escape>	Escapes edit mode subcommands (interpreter only).
<tab>	Moves print position to next tab stop. Tab stops are set every eight columns.
<linefeed>	Moves to next physical line (interpreter only).
<carriage return>	Terminates input of a line.

3.5.2 Control Characters

GW-BASIC supports the following control characters:

Control Character	Action
Control-A	Enters edit mode on the line being typed (interpreter only).
Control-C	With the interpreter, interrupts program execution and returns to BASIC command level. With the compiler, returns to the operating system level if the /D (debug) switch is active.
Control-G	Rings the bell at the terminal.
Control-H	Backspaces. Deletes the last character typed.
Control-I	Tabs to the next tab stop. Tab stops are set every eight columns.
Control-O	Halts program output while execution continues. A second Control-O resumes output.
Control-R	Lists the line that is currently being typed.
Control-S	Suspends program execution (interpreter only).
Control-Q	Resumes program execution after a Control-S (interpreter only).
Control-U	Deletes the line that is currently being typed.

3.6 CONSTANTS

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

3.6.1 String 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"

Numeric constants are positive or negative numbers. GW-BASIC numeric constants cannot contain commas. There are five types of numeric constants:

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

2. Fixed-point constants — Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating-point constants — Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constants is 10-38 to 10+38.

   Examples:

   235.988E-7 = .0000235988
   2359E6 = 2359000000

   (Double precision floating-point constants are denoted by the letter D instead of E. See Section 3.7.2.)

4. Hex constants

Hexadecimal numbers, denoted by the prefix &H.

Examples:

```
&H76
&H32F
```

5. Octal constants

Octal numbers, denoted by the prefix &O or &.

Examples:

```
&O347
&1234
```

6. Binary constants

Binary numbers, denoted by the prefix &B.

Examples:

```
&B123
&B47
```

3.6.2 Single/Double Precision Form For Numeric Constants

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with 7 digits of precision, and printed with up to 6 digits of precision. Double precision numeric constants are stored with 16 digits of precision and printed with up to 16 digits.

A single precision constant is any numeric constant that has one of the following characteristics:

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 has one of these characteristics:

1. Eight or more digits.

2. Exponential form using D.

3. A trailing number sign (#).

Examples:

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

3.7 VARIABLES

Variables are names used to represent values used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero (or null for a string variable).

3.7.1 Variable Names and Declaration Characters

GW-BASIC variable names may be any length. Up to 40 characters are significant. Variable names can contain letters, numbers, and the decimal point. However, the first character must be a letter. Special type declaration characters (listed below) are also allowed.

A variable name may not be a reserved word, but embedded reserved words are allowed. Reserved words include all GW-BASIC commands, statements, function names, and operator names. If a variable begins with FN, it is assumed to be a call too a user-defined function.

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

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

%	Integer variable
!	Single precision variable
#	Double precision variable

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

With the GW-BASIC Compiler, we recommend that you use integer variables whenever possible. Integer variables produce the fastest and most compact object code. For example, the following program executes approximately 30 times faster when the loop control variable "I" is replaced with "I%", or when I is declared an integer variable with DEFINT.

```
FOR I=1 TO 10
A(I)=0
NEXT I
```

Examples of GW-BASIC 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 including the GW-BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL in a program. These statements are described in detail in Section 4.30.

NOTE: With the interpreter, loop control variables must be single precision. With the compiler, however, they may be either single or double precision. Double precision loop control variables let you increase the precision of the increment or increase the range of the loop.

3.7.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32,767.

3.7.3 Space Requirements

The following list gives only the number of bytes occupied by the values represented by the variable names. Additional requirements may vary according to implementation.

Variables

Type	Bytes
Integer	2
Single precision	4
Double precision	8

Arrays

Type	Bytes
Integer	2 per element
Single precision	4 per element
Double precision	8 per element

The compiler and interpreter differ in their implementations and maintenance of string space. Most implementations of the interpreter support strings of up to 255 characters. The number of bytes required for the string descriptor varies with the implementation. With the compiler, strings of up to 32767 characters are supported, and the string descriptor requires 4 bytes of memory.

NOTE: With the compiler, using either POKE with PEEK and VARPTR, or using assembly language subroutines to change string descriptors may cause a "String Space Corrupt" error.

3.8 TYPE CONVERSION

When necessary, GW-BASIC will convert 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 will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

Example:

```
10 A% =23.42
20 PRINT A%
will yield
 23
```

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

   Examples:

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

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

```
10 D=6#/7
20 PRINT D
will yield
 .857143
```

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

3. Logical operators (see Section 3.10.3) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

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

   Example:

```
10 C% =55.88
20 PRINT C%
will yield
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits (rounded) of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value.

   Example:

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

3.9 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. GW-BASIC operators may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

Each category is described in the following sections.

3.9.1 Arithmetic Operators

The arithmetic operators, in order of evaluation, are:

Operator	Operation	Sample Expression
∧	Exponentiation	X∧Y
−	Negation	−X
*,/	Multiplication, Floating-point Division	X*Y X/Y
\	Integer division	12\6=2
MOD	Modulus arithmetic	10.4 MOD 4=2 (10/4=2 with remainder 2)
+,−	Addition, Subtraction	X+Y

With the interpreter, 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. With the compiler, however, parentheses will not always redirect the order of evaluation.

Note the additional differences between the interpreter and compiler:

1. Numeric calculations involving numbers with a large number of decimal places may not produce exactly the same results with the interpreter as with the compiler. This difference affects only calculations involving very precise numbers.

2. During expression evaluation, the GW-BASIC Compiler converts operands of difference types to the type of the more precise operand.

   For instance, the following expression causes J% to be converted to single precision and added to A!:

   ```
   QR=J%+A!+Q#
   ```

   The resultant sum is then converted to double precision and added to Q#.

3. The interpreter always performs transcendental functions in single precision. The compiler performs them in double precision if requested.

The following list gives some sample algebraic expressions and their GW-BASIC counterparts.

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

3.9.1.1 Integer Division and Modulus Arithmetic

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

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

Examples:

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

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

Examples:

```
10.4 MOD 4=2 (10/4=2 with a remainder 2)
25.68 MOD 6.99=5 (26/7=3 with a remainder 5)
```

3.9.1.2 Overflow and Division by Zero

With the interpreter, if division by zero is encountered during the evaluation of an expression, a "Division by zero" error message displayed. Machine infinity (the largest number than can be represented in floating-point format) with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation operator results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the interpreter displays an "Overflow" error message, supplies machine infinity with the algebraically correct sign as the result, and continues execution.

The compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter, the following statements yield 40000 for A%.

```
I% =20000
J% =20000
A% =I% +J%
```

That is, J% is added to I%. Because the number is too large for an integer representation, the interpreter converts the result into a floating-point number. The result (40000) is found and converted back to an integer and saved as A%.

The GW-BASIC Compiler, however, must make type conversion decisions during compilation. It cannot defer until actual values are known. Thus, the compiler generates code to perform the entire operation in integer mode, and arithmetic overflow occurs. If the /D (Debug) switch is set, the error is detected. Otherwise, an incorrect answer is produced.

When the above example is executed with the compiler, I% +J% yields the integer value -25536. This value is then converted to a floating-point value and saved in A%.

Besides these type conversion decisions, the compiler performs certain valid optimizing algebraic transformations before generating code. For example, the following program could produce an incorrect result when run:

```
I% =20000
J% =-18000
```

```
K%=20000
M%=I%+J%+K%
```

If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I%+K% first and then adds J%, overflow does occur.

The compiler follows the rules of operator evaluation, but no other guarantee of evaluation order can be made; even the use of parentheses may not always direct the order of evaluation.

3.9.2 Relational Operators

Relational operators are used to compare two values. 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 IF statements, Section 4.53.)

The relational operators are:

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, Section 4.67.)

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 SIN(X)<0 GOTO 1000
IF I MOD J<>0 THEN K=K+1
```

3.9.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator performs bit-by-bit calculation and returns a result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in Table 3-1. The operators are listed in order of precedence.

Operator	X	Y	Result
NOT			
	X	NOT X	
	1	0	
	0	1	
AND			
	X	Y	X AND Y
	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR			
	X	Y	X OR Y
	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR			
	X	Y	X XOR Y
	1	1	0
	1	0	1
	0	1	1
	0	0	0
EQV			
	X	Y	X EQV Y
	1	1	1
	1	0	0
	0	1	0
	0	0	1
IMP			
	X	Y	X IMP Y
	1	1	1
	1	0	0
	0	1	1
	0	0	1

Table 3-1 GW-BASIC Relational Operators Truth Table

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 IF statements, Section 4.53).

Example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50

IF NOT P THEN 100
```

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

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator 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 will help demonstrate how the logical operators work.

63 AND 16 = 16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16.
15 AND 14 = 14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110).
-1 AND 8 = 8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8.
4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110).
10 OR 10 = 10	10 = binary 1010, so 1010 OR 1010 = 1010 (decimal 10).
-1 OR -2 = -1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.

NOT X = -(X+1)	The two's complement of any integer is the bit complement plus one.

3.9.4 Functional Operators

When a function is used in an expression, it calls a predetermined operation that is to be performed on an operand. GW-BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All GW-BASIC intrinsic functions are described in Chapter 4.

GW-BASIC also allows "user-defined" functions that are written by the programmer. See "DEF FN Statement," Section 4.29.

3.9.5 String Operators

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

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

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

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

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

Examples:

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

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

3.10 ERROR MESSAGES

If an error causes program execution to terminate, an error message is printed. For a complete list of GW-BASIC error codes and error messages, see Appendix A.

GW-BASIC Commands, Statements, and Functions

GW-BASIC commands and statements are described in this chapter. Briefly, these elements can be defined as:

Command — An instruction that returns control to the operating system after the instruction has been performed. LIST and MERGE, for example, are commands. *Commands are used only with the GW-BASIC Interpreter; they are not supported by the GW-BASIC Compiler.*

Statement — An instruction that is entered as part of a program source line. For example, LET and LINE 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 GW-BASIC. These functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the syntax given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions.
I and J	Represent integer expressions.
X$ and Y$	Represent string expressions.

If a floating-point value is supplied where an integer is required, GW-BASIC will round the fractional portion and use the resulting integer.

Note that with GW-BASIC Interpreter, only integer and single precision results are returned by functions. Double precision functions are supported only by the GW-BASIC Compiler.

See Appendix B for information about mathematical functions that are not intrinsic to GW-BASIC.

Each description in this chapter is formatted as follows:

Syntax — Shows the correct syntax for the instruction or function. See the introduction to this manual for syntax notation.

When the term "filespec" is used as an option in the syntax, it refers to a combination of device name and filename, in the correct format for the operating system.

Purpose — Tells what the instruction or function is used for.

Remarks — Describes in detail how the instruction or function is used.

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

In some of the examples in this chapter, interpreter commands are included so that results can be shown more clearly. Though these commands would not be used with the GW-BASIC Compiler, the results of the statement or function would be the same.

Note — Describes special cases or provides additional pertinent information.

GW-BASIC Compiler	Describes ways in which the instruction or function differs between the GW-BASIC Compiler and GW-BASIC Interpreter. If this section is not present, the usage is the same for the interpreter and compiler.

4.1 ABS FUNCTION

Syntax ABS(X)

Purpose To return the absolute value of the expression X.

Example

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

will yield

```
 35
```

4.2 ASC FUNCTION

Syntax ASC(X$)

Purpose To return a numerical value that is the ASCII code for the first character of the string X$. (See Appendix C for ASCII codes.)

Remarks If X$ is null, an "Illegal function call" error is returned.

Example

```
10 X$="TEST"
20 PRINT ASC(X$)
will yield
 84
```

See the CHR$ function, Section 4.12, for details on ASCII-to-string conversion.

4.3 ATN FUNCTION

Syntax ATN(X)

Purpose To return the arctangent of X, where X is in radians. Result is in the range -pi/2 to pi/2 radians.

Remarks The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example

```
10 INPUT X
20 PRINT ATN(X)
will yield
? 3
 1.249046
```

4.4 AUTO COMMAND

Syntax

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

Purpose

To automatically generate line numbers during program entry.

Remarks

AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the existing line and generate the next line number.

If the cursor is moved to another line on the screen, numbering will resume there.

AUTO is terminated by typing CONTROL-C. The line in which CONTROL-C is typed will not be saved. After CONTROL-C is typed, GW-BASIC returns to command level.

Example

AUTO 100,50

Generates line numbers 100, 150, 200 . . . .

AUTO

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

GW-BASIC Compiler

The AUTO command is not supported by the GW-BASIC Compiler.

4.5 BEEP STATEMENT

Syntax BEEP

Purpose Sounds the speaker at 830 Hz for 240ms.

Remarks Non-graphic versions of MS-BASIC use PRINT CHR$ to send an ASCII Bell Character.

Example 2430 IF X < 20 THEN BEEP 'X is out or range, complain.

4.6 BLOAD STATEMENT

Syntax

BLOAD <filespec> [,<offset>]

The device designation portion of the filespec is optional. The filename may be 1 to 8 characters long.

<offset> is a numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset address at which loading is to start in the segment declared by the last DEF SEG statement.

Purpose

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

Remarks

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 "BSAVE Statement," Section 4.7, for information about saving memory image files.

If the offset is omitted, the segment address and offset contained in the file (i.e., 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.

If offset is specified, the segment address used is the one given in the most recently executed DEF SEG statement. If no DEF SEG statement has been given, the GW-BASIC data segment will be used as the default (because it is the default for DEF SEG).

CAUTION: **BLOAD does not perform an address range check. It is therefore possible to load a file anywhere in memory. The user must be careful not to load over**

GW-BASIC or the operating system.

Example

```
10 'Load subroutine at 60:F000
20 DEF SEG=&H6000 'Set segment at 6000
Hex
30 BLOAD"PROG1",&HF000 'Load PROG1
```

This example sets the segment address at 6000 Hex and loads PROG1 at F000.

GW-BASIC Compiler

The BLOAD statement is not supported by the GW-BASIC Compiler.

4.7 BSAVE STATEMENT

Syntax

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

The device designation portion of the filespec is optional. The filename may be 1 to 8 characters long.

<offset> is a numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset address to start saving from in the segment declared by the last DEF SEG statement.

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

Purpose

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

Remarks

The <filespec>, <offset>, and <length> are required in the syntax.

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

If the offset is omitted, a "Bad file name" error is issued and the save is aborted. A DEF SEG statement must be executed before the BSAVE. The last known DEF SEG address will be used for the save.

If length is omitted, a "Bad file name" error is issued and the save is aborted.

Example

```
10 'Save PROG1
20 DEF SEG=$H6000
30 BSAVE"PROG1",&HF000,256
```

This example saves 256 bytes starting at 6000:F000 in the file PROG1.

GW-BASIC Compiler

The BSAVE statement is not supported by the GW-BASIC Compiler.

4.8 CALL STATEMENT

Syntax

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

where <variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name.

<argument list> contains the arguments that are passed to the external subroutine. <argument list> may contain only variables.

Purpose

To call an assembly language subroutine or a compiled routine written in another high level language.

Remarks

The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function, Section 4.140.)

The CALL statement generates the same calling sequence used by Microsoft FORTRAN and Microsoft BASIC compilers.

Example

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
 .
 .
 .
```

GW-BASIC Compiler

In a compiled GW-BASIC program, line 110 in the above example is not required because the address of MYROUT will be assigned by the linking loader at load time.

Additional differences for GW-BASIC Compiler are:

1. The <variable name> is the name of the subroutine that is to be called. The name must be 1 to 31 characters long and must be recognized by MS-LINK as a global symbol.

That is, <variable name> must be a PUBLIC symbol in an assembly language routine.

2. Since GW-BASIC Compiler allows strings to be up to 32767 bytes long, the string descriptor requires four bytes rather than three as in the interpreter. The four bytes are: low byte, high byte of the length, followed by low byte, high byte of the address. If the assembly language routine uses string arguments, it may need to be recoded to account for this difference. (See the *NCR GW-BASIC Compiler User's Guide*, Chapter 9.)

4.9 CALLS STATEMENT

The CALLS statement is just like CALL, except that the segmented addresses of all arguments are passed. (CALL passes unsegmented addresses.) CALLS should be used when accessing routines written with FORTRAN calling conventions, since all FORTRAN parameters are call-by-reference segmented addresses.

With the interpreter only, CALLS uses the segment address defined by the most recently executed DEF SEG statement to locate the routine being called.

4.10 CDBL FUNCTION

Syntax CDBL(X)

Purpose To convert X to a double precision number.

Example

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

will yield

```
 454.67 454.6699829101563
```

4.11 CHAIN STATEMENT

Syntax

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

See the examples below for illustration of the syntax options.

Purpose

To call a program and pass variables to it from the current program.

Remarks

<filespec> is the spec of the program that is called.

The COMMON statement may be used to pass variables (see Section 4.20).

<line number exp> 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. <line number exp> is not affected by a RENUM command.

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 list the variables that are passed. See Section 4.20 for information about COMMON.

If the ALL option is used and <line number expression> is not, a comma must hold the place of <line number expression>. For example, CHAIN "NEXTPROG",,ALL is correct; CHAIN "NEXTPROG",ALL is incorrect. In the latter case, GW-BASIC assumes that ALL is a variable name and evaluates it as a line number expression.

The MERGE option allows a subroutine to be brought into the GW-BASIC program as an overlay. That is, the current program and the called program are merged (see "MERGE

Command," Section 4.80). The called program must be an ASCII file if it is to be merged.

After an overlay is used, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option.

The line numbers in <range> are affected by the RENUM command.

Example 1

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

Example 2

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

Example 3

```
10 REM THIS PROGRAM DEMONSTRATES
CHAINING USING THE MERGE, ALL, AND
DELETE OPTIONS.
```

```
20 REM SAVE THIS MODULE ON THE DISK
AS "MAINPRG".
30 A$="MAINPRG"
40 CHAIN MERGE "OVRLAY1",1010,ALL
50 END

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

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

Note

The CHAIN statement with MERGE option leaves the 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 DEFFN 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.

GW-BASIC Compiler

The GW-BASIC Compiler does not support the ALL, MERGE, DELETE, and <line number exp> options to CHAIN. Thus, the statement

syntax is CHAIN <filespec>. If you wish to maintain compatibility with GW-BASIC Compiler, we recommend that you use COMMON to pass variables and that you do not use overlays. The CHAIN statement leaves the files open during chaining.

See the "GW-BASIC Compiler" portion of the COMMON statement, Section 4.20, for more information about chaining with COMMON.

4.12 CHR$ FUNCTION

Syntax

CHR$(I)

Purpose

To return a string whose one character is ASCII character I. (ASCII codes are listed in Appendix C.)

Remarks

CHR$ is commonly used to send a special character to the terminal. For instance, the BELL character (CHR$(7)) could be sent as a preface to an error message, or a form feed (CHR$(12)) could be sent to clear a terminal screen and return the cursor to the home position.

Example

```
PRINT CHR$(66)
```

will yield

```
B
```

See the ASC function, Section 4.2, for details on ASCII-to-numeric conversion.

4.13 CINT FUNCTION

Syntax CINT(X)

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

Remarks If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Example PRINT CINT(45.67)
will yield
 46

See the CDBL and CSNG functions for details on converting numbers to the double precision and single precision data type, respectively. See also the FIX and INT functions, both of which return integers.

4.14 CIRCLE STATEMENT

Syntax

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

Purpose

Draws an ellipse according to the following definitions:

x,y
Specifies the coordinates of the center of the ellipse.

radius
Specifies the radius (major axis) in points.

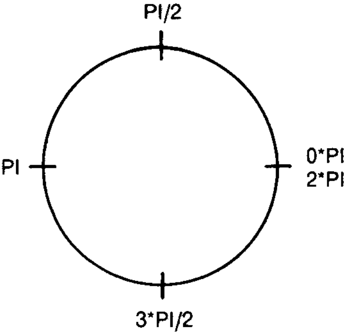
color
Specifies the color of the circle (0-7, see Color statement). If not specified, the color is the foreground color.

start,end
Specifies in radians where the drawing is to begin and end. The values may range from -2*PI to 2*PI, where PI = 3.141593. (See also remarks.)

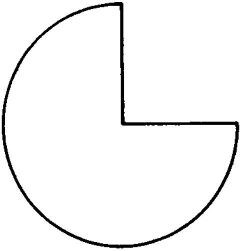
aspect
Specifies the ratio of the X radius to the Y radius. (If not specified, the ratio is assumed to be 1/1, a circle.) If the ratio is less than 1, the radius is the X radius; if the ratio is greater than 1, the radius is the Y radius.

Remarks

The first two arguments (x,y coordinates and radius) are the only ones required to draw a circle. Use the last two arguments to draw other "curved" shapes. Start and end, for example, allow you to control how much of the circle is to be drawn. The values of start and end are in radians, positioned in the standard mathematical way.



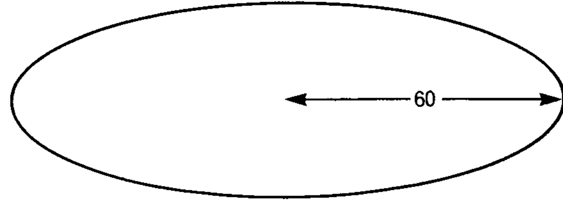
Either start or end value may be negative (-0, however, is not allowed) in which case the angle is connected to the center point with a line. For example, start and end values of -PI/2, -2PI would draw part of a circle.



Use the aspect argument to draw an ellipse other than a circle. If the aspect ratio is less than 1, then r is the X radius; if the aspect ratio is greater than 1, then r is the Y radius. For example,

```
10 SCREEN 1
20 CIRCLE (160,100),60,,,,5/18
```

will draw an ellipse like this:



NCR Corporation is pleased to provide GW-BASIC software for implementation on your NCR Decision Mate V. Your GW-BASIC package contains an *NCR GW-BASIC Reference Manual*, a *GW-BASIC User's Guide* for either the GW-BASIC Interpreter or the GW-BASIC Compiler, and a disk which holds the following files:

NCR GW-BASIC (Interpreter)

For MS™-DOS
Disk 1 of 1

GWBASIC.EXE
GWCONF.COM
DUMPCL.OBJ

NCR GW-BASIC Compiler

For MS™-DOS
Disk 1 of 1

GWBCOM.COM
BASCOMG.LIB
BASRUNG.LIB
BASRUNG.EXE
GWCONF.COM
DEMO.BAS
LINK.EXE

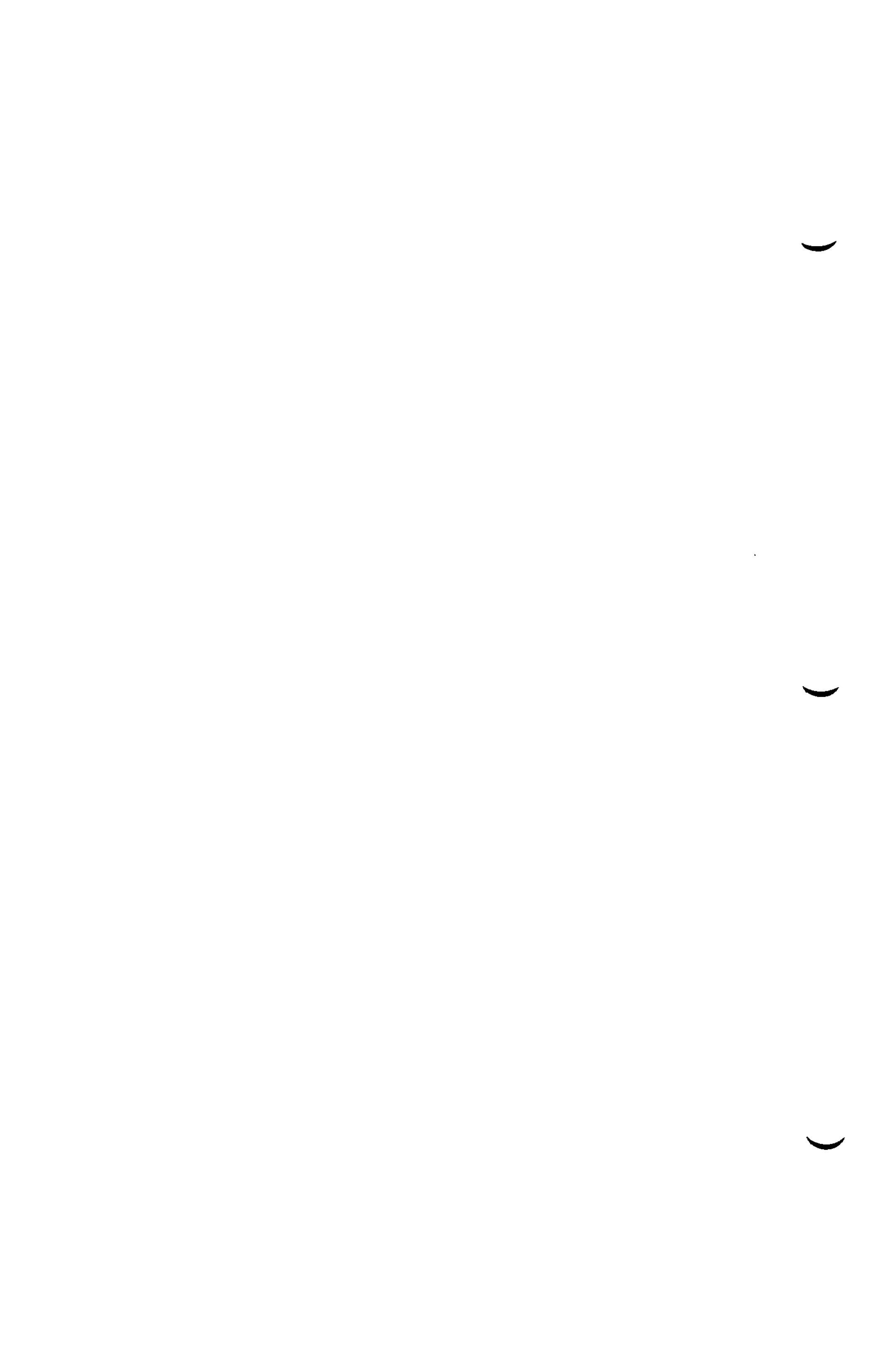
NCR GW-BASIC Compiler

The GW-BASIC Compiler program has been pre-installed for your NCR Decision Mate V.

No programmable function key assignments have been made. To define your own, see the KEY Statement in Chapter 4, Section 4.61, of your *NCR GW-BASIC Reference Manual.*

F1	
F2	
F3	
F4	
F5	
F6	
F7	
F8	
F9	
F10	
F11	
F12	
F13	
F14	
F15	

F16	
F17	
F18	
F19	
F20	

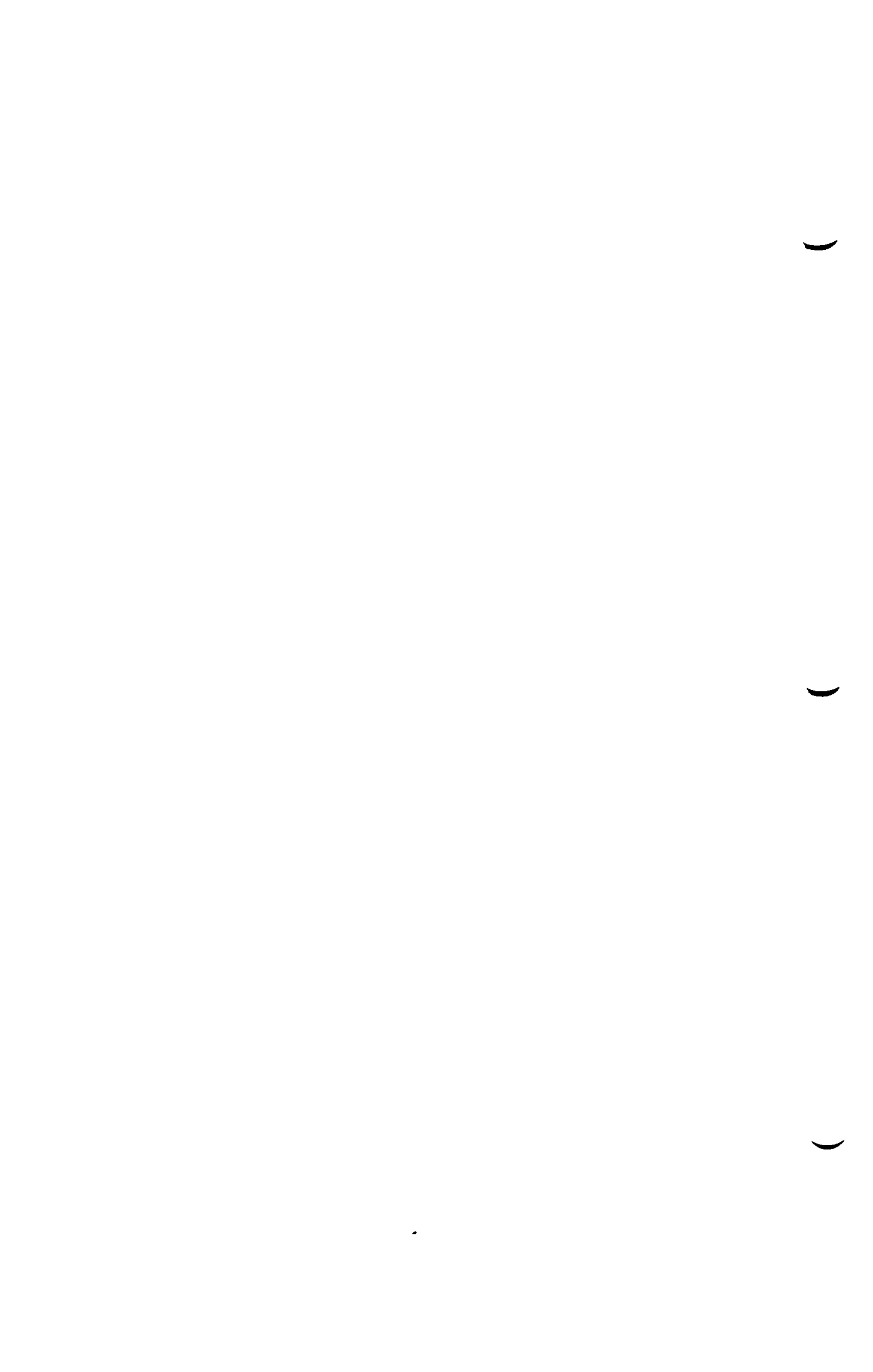


NCR GW-BASIC (Interpreter)

The GW-BASIC program has been pre-installed for your NCR Decision Mate V. The programmable function keys have been assigned the values which appear below. See the KEY Statement in Chapter 4, Section 4.61 of your *GW-BASIC REFERENCE MANUAL* for detailed instructions in utilizing these function keys.

F1	LOAD
F2	RUN
F3	CONT
F4	SAVE
F5	LIST
F6	EDIT
F7	TRON
F8	TROFF
F9	PRINT
F10	PRINT USING
F11	GOTO
F12	GOSUB
F13	IF
F14	THEN
F15	ELSE

F16	CHR$
F17	STRING$
F18	LINE
F19	CIRCLE
F20	DRAW



GW™-BASIC Reference Manual

For MS™-DOS

COPYRIGHT NOTICE

Copyright© 1983 by Microsoft Corporation, all rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a 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 the prior written permission of Microsoft Corporation.

TRADEMARKS

Microsoft and the Microsoft logo are registered trademarks of Microsoft Corporation. MS, GW, Music Macro Language, and Graphics Macro Language are trademarks of Microsoft Corporation. Teletype is a registered trademark of Teletype Corporation.

DISCLAIMER OF WARRANTY

NCR Corporation and Microsoft Corporation make no representations or warranties with respect to the contents hereof and specifically disclaim any implied warranties of merchantability or fitness for any particular purpose. Further, NCR Corporation and Microsoft Corporation reserve the right to revise this publication and to make changes from time to time in the content hereof without obligation to notify any person or organization of such revisions or changes.

The GW-BASIC Compiler Software and Manual are sold AS IS and without warranty as to performance. While NCR Corporation and Microsoft Corporation firmly believe this to be a high quality product, the user must assume all risks of using the program.

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

4.15 CLEAR STATEMENT

Syntax

CLEAR [,[<expression1>][,<expression2>]]

Purpose

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

Remarks

<expression1> is a memory location that, if specified, sets the highest location available for use by GW-BASIC.

<expression2> sets aside stack space for GW-BASIC. The default is 768 bytes or one-eighth of the available memory, whichever is smaller.

Note

GW-BASIC allocates string space dynamically. An "Out of string space" error occurs only if there is no free memory left for GW-BASIC to use.

The CLEAR statement performs the following actions:

Closes all files.
Clears all COMMON variables.
Resets numeric variables and arrays to zero.
Resets the stack and string space.
Resets all string variables and arrays to null.
Releases all disk buffers.
Resets all DEF FN and DEF/SNG/DBL/STR statements (for interpreter only).

Examples

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000
```

GW-BASIC Compiler

GW-BASIC Compiler supports the CLEAR statement with the restriction that <expression1> and <expression2> must be integer expressions. If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 768 bytes, and the default top of memory is the current top of memory.

4.16 CLOSE STATEMENT

Syntax

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

Purpose

To conclude I/O to a file. The CLOSE statement is complementary to the OPEN statement.

Remarks

<file number> is the number under which the file was opened. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE statement. The file may then be reopened using the same or a different file number; likewise, that file number may now be reused to open any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always close all disk files automatically. (STOP does not close disk files.)

Example

```
CLOSE #1,#2
```

4.17 CLS STATEMENT

Syntax CLS

Purpose Erases the screen to the currently selected background color.

Remarks You may also clear the screen with the CONTROL-L or CONTROL-HOME (↖) keys. (The SCREEN statement also clears the screen.)

If the KEY ON statement is in effect when you use the CLS statement, the screen is cleared; however, the function line at the bottom of the screen is renewed with the currently active background/foreground colors.

4.18 COLOR STATEMENT

Syntax

COLOR [foreground] [,background]

foreground
Specifies the character color. Enter an unsigned integer in the range 0-7 (see following chart).

background
Specifies the screen color. Enter an unsigned integer in the range 0-7 (see following chart).

Purpose

Changes either (or both) the foreground or background colors. The colors are specified by codes:

0 = black
1 = blue
2 = green
3 = cyan
4 = red
5 = magenta
6 = yellow
7 = white

Remarks

You may omit either parameter, in which case the value from a previous COLOR statement (if any) is assumed.

The foreground and background colors may be the same, making the characters invisible.

If converting programs, note that NCR GW-BASIC accepts a third parameter and codes 8-31 without displaying an error message. Also, the syntax does not differ for text and graphics mode.

An illegal parameter value results in an "Illegal Function Call" message. The screen colors remain as they were before the statement was entered.

Examples

10 COLOR 4,7 Uses red characters on a white background.

20 COLOR ,,4 Changes the background to red; character are invisible.

30 COLOR 2,0 Uses green characters on a black background.

4.19 COM STATEMENT

Syntax

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

where (n) is the number of the communications channel. The range for (n) is specified by the implementor.

Purpose

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

The COM(n) ON statement enables communications event trapping by an ON COM statement (see "ON COM Statement," Section 4.87). While trapping is enabled, and if a non-zero line number is specified in the ON COM statement, GW-BASIC checks between every statement to see if activity has occurred on the communications channel. If it has, the ON COM statement is executed.

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

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

Note

For additional information on communications event trapping, see "Event Trapping," Section 1.6, and "ON COM Statement," Section 4.87.

Example

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

Enables error trapping of communications activity on channel 1.

GW-BASIC Compiler

See compiler note under "ON COM Statement," Section 4.87.

4.20 COMMON STATEMENT

Syntax

COMMON <list of variables>

Purpose

To pass variables to a chained program.

Remarks

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

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

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

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

Example

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

GW-BASIC Compiler

With the compiler, the COMMON statement must appear in a program before any executable statements. The current nonexecutable statements are:

COMMON
DEFDBL, DEFINT, DEFSNG, DEFSTR
DIM
OPTION BASE
REM
$INCLUDE (see GW-BASIC Operations Guide)

Array variables used in a COMMON statement must be declared in a preceding DIM statement.

The standard form of the COMMON statement is referred to as "blank" COMMON. The GW-BASIC Compiler also supports Microsoft FORTRAN Compiler-style "named" COMMON areas; however, the variables are not preserved across chains. The syntax for named COMMON is:

COMMON /<name>/ <list of variables>

where <name> consists of 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with programs that use FORTRAN calling conventions and assembly language routines, without having to explicitly pass parameters in the CALL statement.

With the compiler, the order of variables must be the same for all COMMON statements communicating between chaining and chained-to programs. If the size of the common region in the chained-to program is smaller than the region in the chaining program, the extra COMMON variables in the chaining program are ignored. If the size of the common region in the chained-to program is larger, the additional COMMON variables are initialized to zeros and null strings.

To ensure that common areas can be shared between programs, place COMMON declarations in a single include file and use the $INCLUDE statement in each program. (See the *NCR GW-BASIC User's Guide* for discussion of the $INCLUDE statement.) For example:

```
10 REM This file is MENU.BAS
20 REM $INCLUDE:'COMDEF'
 .
 .
 .
1000 CHAIN "PROG1"

10 REM This file is PROG1.BAS
20 REM $INCLUDE:'COMDEF.BAS'
 .
 .
 .
2000 CHAIN "MENU"

10 REM This file is COMDEF.BAS
100 DIM A(100),B$(200)
110 COMMON I,J,K,A()
120 COMMON A$,B$(),X,Y,Z
130 REM End COMDEF.BAS
```

The BASCOMG.LIB runtime library does not support COMMON with chained programs. Therefore, programs should not be compiled with the /O switch if they use the COMMON statement in conjunction with the CHAIN statement.

4.21 CONT COMMAND

Syntax

CONT

Purpose

To continue program execution after a Control-C has been typed or a STOP or END statement has been executed.

Remarks

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 STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error has occurred.

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

Example

See "STOP Statement," Section 4.129.

GW-BASIC Compiler

The CONT command is not supported by the GW-BASIC Compiler.

4.22 COS FUNCTION

Syntax COS(X)

Purpose To return the cosine of X, where X is in radians.

Remarks The calculation of COS(X) is performed in single precision.

Example

```
10 X=2*COS(.4)
20 PRINT X
will yield
 1.842122
```

4.23 CSNG FUNCTION

Syntax CSNG(X)

Purpose To convert X to a single precision number.

Example

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

will yield

```
 975.3421 975.3421
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types, respectively.

4.24 CSRLIN FUNCTION

Syntax

X = CSRLIN

X
Specifies any numeric variable for which the software returns a value in the range 1 through 24.

Purpose

Returns the current line position of the cursor.

Example

In the following example, the statement in line 10 returns the current line position. In line 20 the statement returns the current column position; in line 30 it prints HELLO in the middle of the screen, and in line 40 it restores the position of the cursor to the previous line and column. The software returns a value for X = POS(O) in the range 1 through 80.

```
10 Y = CSRLIN
20 X = POS(O)
30 LOCATE 12,40 :PRINT "HELLO"
40 LOCATE Y,X
```

4.25 CVI, CVS, CVD FUNCTIONS

Syntax

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

Purpose

To convert string values to numeric values.

Remarks

Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example

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

See also "MKI$, MKS$, MKD$ Functions," Section 4.83.

4.26 DATA STATEMENT

Syntax

DATA <list of constants>

Purpose

To store the numeric and string constants that are accessed by the program's READ statement(s). (See "READ Statement," Section 4.109.)

Remarks

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

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

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

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 4.113).

Example

See "READ Statement," Section 4.109.

4.27 DATE$ STATEMENT

Syntax

DATE$=<string expression>

<string expression> returns a string in one of the following forms:

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

Purpose

To set the current date. This statement complements the DATE$ function, which retrieves the current date.

Example

```
10 DATE$="01-15-1984"
```

The current date is set at January 15, 1984.

4.28 DATE$ FUNCTION

Syntax x$=DATE$

Purpose To retrieve the current date. (To set the date, use the DATE$ statement, described in Section 4.27.)

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

Example

```
10 PRINT DATE$
```

The DATE$ function prints the date, calculated from the date set with the DATE$ statement.

4.29 DEF FN STATEMENT

Syntax DEF FN<name>[(<parameter list>)]=
<function definition>

Purpose To define and name a function that is written by the user.

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

<parameter list> consists of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

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

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

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

A DEF FN statement must be encountered before the function it defines may be called. If a function is called before it has been defined, an

"Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example

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

Line 410 defines the function FNAB. The function is called in line 420.

4.30 DEFINT/SNG/DBL/STR STATEMENTS

Syntax

DEF<type> <range(s) of letters>

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

Purpose

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

Remarks

Any variable names beginning with the letter(s) specified in <range of letters> will be considered the type of variable specified in the <type> portion of the statement. However, a type declaration character always takes precedence over a DEFtype statement. (See "Variable Names and Declaration Characters," Section 3.7.1.)

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

Examples

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

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

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

GW-BASIC Compiler

The compiler does not "execute" a DEFxxx statement, as it does a PRINT statement, for example. A DEFxxx statement takes effect as soon as it is encountered in the program during compilation. Once the type has been defined for the listed variables, that type remains in effect either until the end of the program or until

another DEFxxx statement alters the type of the variable. Unlike the interpreter, the compiler cannot circumvent the DEFxxx statement by directing flow of control around it with a GOTO statement. For variables given with a precision designator (i.e., %, !, #, as in A% =B), the type is not affected by the DEFxxx statement.

At compiletime, the compiler allocates memory for storage of designated variables, and assigns them one of the following data types:

1. Integer (INT)

2. Single precision floating-point (SNG)

3. Double precision floating-point (DBL)

4. String (STR)

4.31 DEF SEG STATEMENT

Syntax

DEF SEG [=<address>]

where <address> is a numeric expression returning an unsigned integer in the range 0 to 65535.

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.

Remarks

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

Entry of any value outside the <address> range 0 through 65535 will result in an "Illegal function call" error, and the previous value will be retained.

If the <address> option is omitted, the segment to be used is set to the GW-BASIC data segment (DS). This is the initial default value.

If the <address> option is given, it should be based on a 16-byte boundary. GW-BASIC does not check the validity of the specified address.

Note

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

Example

```
10 DEF SEG=&HB800 'Seg segment to &800 Hex
20 DEF SEG 'Restore segment to GW-BASIC
data segment
```

GW-BASIC Compiler

With the compiler, DEF SEG is referenced only by the POKE statement and the PEEK and USR functions.

4.32 DEF USR STATEMENT

Syntax

DEF USR[<digit>]=<integer expression>

Purpose

To specify the starting address of an assembly language subroutine.

Remarks

<digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine.

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

Example

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

4.33 DELETE COMMAND

Syntax
DELETE [<line number>][-<line number>]

Purpose
To delete program lines.

Remarks
GW-BASIC always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples
DELETE 40
Deletes line 40.

DELETE 40-100
Deletes lines 40 through 100, inclusive.

DELETE -40
Deletes all lines up to and including line 40.

GW-BASIC Compiler
The DELETE command is not supported by the GW-BASIC Compiler.

4.34 DIM STATEMENT

Syntax

DIM <list of subscripted variables>

Purpose

To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks

If an array variable name is used without a DIM statement, the maximum value of the array's subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 4.94).

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

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

Example

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

GW-BASIC Compiler

With the compiler, the DIM statement is scanned rather than executed. That is, DIM takes effect when it is encountered at compiletime and remains in effect until the end of the program. It cannot be reexecuted at runtime.

If the default dimension (10) has already been established for an array variable, and that variable is later encountered in a DIM statement, an "Array already dimensioned" error results. Therefore, the practice of putting a collection of DIM statements in a subroutine at the end of a program generates severe errors. In that case, the compiler sees the DIM statement only after it has already assigned the default dimension to arrays declared earlier in the program.

The values of the subscripts in a DIM statement must be integer constants; they may not be variables, arithmetic expressions, or floating-point values.

4.35 DRAW STATEMENT

Syntax

DRAW <string expression>

Purpose

Draws an object as specified by the string expression.

Remarks

With the Draw statement you can draw an object using object definition language commands. A language command is a single character within a string, optionally followed by one or more arguments. The string expression defines an object which is drawn on the screen when BASIC executes the DRAW statement.

The following movement commands begin movement from the coordinates of the last point plotted with another language command, LINE statement, or PSET statement. When a program is RUN, movement begins from the center of the screen (320,200).

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 down and right
G [<n>] Move diagonally down and left
H [<n>] Move diagonally up and left

The n in the preceding commands indicates the distance to move. The number of points moved is n times the scale factor (see S below). If you do not specify n, commands move one unit.

M<x,y>
Move absolute or offset (see Chapter 1 for discussion of x and y coordinates). If x is preceded by a + or -, x and y are added to the coordinates of the last point plotted and connected to the current point by a line. If no + or - is added, a line is drawn to point (x,y) from the current point.

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

B
Move but do not plot any points.

N
Move but return to original position when done.

A <n>
Set angle n. n may range from 0 to 3, where 0 is 0 degrees, 1 is 90 degrees, 2 is 180 degrees, and 3 is 270 degrees.

C <n>
Set color n. n may range from 0 to 7 (see Color statement).

S <n>
Set scale factor. n may range from 1 to 255. The scale factor multiplied by the distances given with the U, D, L, R, E, F, G, H, and M commands gives the actual distance moved.

X <string>
Execute substring. Allows you to execute a second substring from a string, much like GOSUB in BASIC. Arguments can be constants like 123 or =variable, where variable is the name of a variable.

Examples

To draw a triangle:

```
10 SCREEN 1
20 DRAW "E60;F60;L120"
```

To draw a box:

```
10 SCREEN 1
20 V = 100
30 DRAW "U=V;R=V;D=V;L=V"
```

GW-BASIC Compiler

The GW-BASIC Compiler does not support the X <string expression> subcommand. However, you can execute a substring by appending the character form of the address to "X". For example, the following two statements are equivalent. The first statement would be used with the interpreter, the second with the compiler.

```
DRAW "XA$;"

DRAW "X"+VARPTR$(A$)
```

4.36 EDIT COMMAND

Syntax

EDIT <line number>

line number
Specifies the line number of a line in the program. If there is no such line, an "Undefined Line Number" error message is displayed.

Purpose

Displays a line for editing.

Remarks

The EDIT statement simply displays the line specified and positions the cursor under the first digit of the line number. You may then modify the line using the keys described in the Full Screen Editor section of Chapter 2.

A period (.) always refers to the current line. If you have just entered a line and want to go back and edit it, you may enter EDIT. to redisplay the line.

GW-BASIC Compiler

The EDIT command is not supported by the GW-BASIC Compiler.

4.37 END STATEMENT

Syntax END

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

Remarks 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 printed. An END statement at the end of a program is optional. GW-BASIC always returns to command level after an END is executed.

Example

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

4.38 EOF FUNCTION

The EOF function may be used with random access files as well as sequential files.

Syntax

EOF(<file number>)

Purpose

To test for the end-of-file condition.

For sequential files, the EOF function returns true (-1) if there is no more data in the file. The file is empty if the next input operation (INPUT, LINE INPUT, INPUT$, for example) would cause an "Input past end" error.

For random access files, the EOF function returns true (-1) if the most recently executed GET statement attempts to read beyond the end-of-file.

Remarks

Because MS-BASIC allocates 128 bytes to a file at a time, it is possible that EOF will not accurately detect the end of a random access file which has been opened with a record length of less than 128 bytes. For example, if a file is opened with a record length of 64 bytes and one record is written to the file (e.g., PUT#1,1), EOF will return false if a GET statement is attempted on the second record of the file (e.g., GET#1,2). This will occur even though this record has not actually been written to.

Example

```
10 REM
20 REM Open the library catalog file,
30 REM LIBRARY.DAT.
40 OPEN "R",#1,"LIBRARY.DAT"
50 REM The first 35 bytes of the
60 REM record contain the title,
70 REM the remaining 93 bytes con-
80 REM tain additional information which
90 REM is not used by this program.
100 FIELD+1,35 AS TITLE$,93 AS G$
110 REM
120 REM Initialize the number of books seen.
130 REM
```

```
140 NBOOKS=0
150 REM Attempt to fetch the next record.
160 REM Note that the record number
170 REM argument of GET isn't specified
180 REM so the next record of the
190 REM file is fetched.
200 GET #1
210 REM
220 REM Is this the end of the file?
230 REM
240 IF EOF(1) THEN 1000
250 REM No, increment the count of books,
260 REM print the current title, and
270 REM loop back to read the next
280 REM record.
290 NBOOKS=NBOOKS+1
300 PRINT TITLE$
310 GOTO 200
1000 REM Here when the end of file has
1010 REM been reached. Print a blank line
1020 REM and the number of books, close the
1030 REM file and terminate the program.
1040 PRINT "These are ";NBOOKS;" books in";
1050 PRINT "your library."
1060 CLOSE
1070 END
```

This sample program lists the titles of the books cataloged in the file LIBRARY.DAT. It also counts the books in the library by counting the number of records which are read from LIBRARY.DAT before the end-of-file is encountered.

Each record of LIBRARY.DAT contains information on one book in the library. The record length is 128 bytes. The first 35 bytes contain the title of the book; the remaining 93 bytes contain additional information about the book (e.f., author, publisher, location, etc.). This information is not used in this example.

4.39 ERASE STATEMENT

Syntax

ERASE <list of array variables>

Purpose

To eliminate arrays from memory.

Remarks

Arrays may be redimensioned after they are erased, or the previously allocated array space in memory may be used or other purposes. If an attempt is made to redimension an array without first erasing it, a "Duplicate definition" error occurs.

Example

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

GW-BASIC Compiler

GW-BASIC Compiler does not support ERASE.

4.40 ERR AND ERL VARIABLES

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

With the GW-BASIC Interpreter, if the statement that caused the error was a direct mode statement, ERL will contain 65535. To test whether an error occurred in a direct statement, use IF 65535=ERL THEN .... Otherwise, use

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

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

If the line number is not on the right side of the relational operator, it cannot be renumbered with RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. GW-BASIC error codes are listed in Appendix A.

4.41 ERROR STATEMENT

Syntax

ERROR <integer expression>

Purpose

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

Remarks

ERROR can be used as a statement (part of a program source line) or as a command (in direct mode).

The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix A), the ERROR statement will simulate the occurrence of that error and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by GW-BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to GW-BASIC.) This user-defined error code may then be conveniently handled in an error handling routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, GW-BASIC responds with the "Unprintable error" error message. Execution of an ERROR statement for which there is no error handling routine causes an error message to be printed and execution to halt.

Example 1

```
10 S=10
20 T=5
30 ERROR S+T
40 END
```

will yield

String too long in line 30

Or, in direct mode (interpreter only):

```
Ok
ERROR 15        (You type this line.)
String too long (GW-BASIC types this line.)
Ok
```

Example 2

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

4.42 EXP FUNCTION

Syntax	EXP(X)
Purpose	To return e (base of natural logarithms) to the power of X. X must be <= 88.02969.
Remarks	If x is greater than 88.02969, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.
Example	10 X=5 20 PRINT EXP(X-1) will yield 54.59815

4.43 FIELD STATEMENT

Syntax

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

Purpose

To allocate space for variables in a random file buffer.

Remarks

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

<file number> is the number under which the file was opened. <field width> is the number of characters to be allocated to <string variable>.

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

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

Note

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

Example 1

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

Allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does not place any data in the random file buffer. (See also "GET Statement," Section 4.48, and "LSET and RSET Statements," Section 4.79.)

Example 2

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

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

Example 3

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

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

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

Example 4

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

```
AS A$(LOOP%)
160 OFFSET%=OFFSET%+SIZE%(LOOP%)
170 NEXT LOOP%
```

Creates a field in the same manner as Example 3. However, the element size varies with each element. The equivalent declaration is:

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

GW-BASIC Compiler

The compiler does not permit fielded strings to be passed in COMMON.

4.44 FILES STATEMENT

Syntax

FILES [<filespec>]

where <filespec> includes a filename and optional device designation.

Purpose

To print the names of files residing on the specified disk.

Remarks

If <filespec> is omitted, all the files on the currently selected drive will be listed. <filespec> is a string formula which may contain question marks (?) or asterisks (*) used as wild cards. A question mark will match any single character in the filename or extension. An asterisk will match one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks.

Examples

FILES

Shows all files on currently logged disk.

FILES "*.BAS"

Shows all files with extension .BAS.

FILES "B:*.*"

Shows all files on drive B.

FILES "B:" (equivalent to "B:*.*")

FILES "TEST?.BAS"

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

4.45 FIX FUNCTION

Syntax FIX(X)

Purpose To return the truncated integer part of X.

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

Examples PRINT FIX(58.75)
will yield
 58

PRINT FIX(-58.75)
will yield
-58

4.46 FOR...NEXT STATEMENT

Syntax

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

where x, y, and z are numeric expressions.

Purpose

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

Remarks

<variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is adjusted by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, GW-BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decreased each time through the loop. 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 will result.

The body of the loop is skipped if the initial value of the loop times the sign of the STEP exceeds the final value times the sign of the STEP.

Nested Loops

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
```

will yield

```
 1   20
 3   30
 5   40
 7   50
 9   60
```

Example 2

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

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

Example 3

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

will yield

```
 1  2  3  4  5  6  7  8  9  10
```

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

GW-BASIC Compiler

Double precision FOR...NEXT loops may e used with the compiler if extra precision is desired.

4.47 FRE FUNCTION

Syntax

FRE(0)
FRE("")

Purpose

With a numeric argument, FRE returns the number of bytes in memory that are not being used by GW-BASIC. Arguments to FRE are dummy arguments.

Remarks

FRE("") forces a garbage collection before returning the number of free bytes. With the interpreter, garbage collection may take 1 to 1 1/2 minutes. It is much faster with the compiler.

GW-BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

Example

PRINT FRE(0)
will yield
 14542

GW-BASIC Compiler

With the compiler, FRE with a numeric argument returns the size of the largest block of free string space. With a string argument, garbage collection is performed as in the interpreter, but FRE returns the amount of available string space only.

4.48 GET STATEMENT

Syntax

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

Purpose

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

Remarks

<file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

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

See "GET and PUT Statements" in this chapter, for discussion of the GET and PUT statements used with screen capabilities.

Example

```
GET #1,75
```

Note

After a GET statement has been executed, INPUT# and LINE INPUT # may be executed to read characters from the random file buffer.

4.49 GET AND PUT STATEMENTS

Syntax

GET (x1,y1)-(x2,y2),<array name>
PUT (x1,y1),<array name>[,<action verb>]

Purpose

The GET and PUT statements transfer graphics images to and from the screen. The statements also make it possible to perform animation and high-speed image motion.

Remarks

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

(x1,y1) and (x2,y2)
Coordinates in absolute or offset form (see Chapter 1) of the opposite corners of a rectangle.

array name
Your name of the array which will hold image information.

GET reads into an array the colors of the points in the screen image bounded by the rectangle. The rectangle is defined the same way as the rectangle drawn by the Line statement using the ",b" option.

The array is used simply as a place to hold the image bounded by the rectangle. It must be numeric and dimensioned large enough to hold the entire image. You may determine the required array size in bytes using the following formula:

$$\mathrm{INT}\frac{(x+7)*}{8}\frac{\text{bits*}}{\text{pixel}}y+4$$

where x is the length of a horizontal side of the rectangle and y is the length of a vertical side of the rectangle. Bits per pixel is 3 in color mode and 1 in black and white mode. The bytes per element of an array are:

2 for integer
4 for single precision

8 for double precision

For example, if you want to use the GET statement to put a 10 by 12 image into an array, the number of bytes required is:

$\mathrm{INT}(\frac{10+7}{8})*3*12+4$ or 76 bytes.

So you would need an integer array $(\frac{76}{2})$ of at least 38 bytes.

The storage format in the array is as follows:

2 bytes giving x dimension in bits
2 bytes giving y dimension in bits
the array data

The data for each row of points is left justified on a byte boundary. If there is less than a multiple of 8 bits stored, the rest of the byte will be filled out with zeroes.

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

(x1,y1)
Coordinates of the top left corner of the image to be transferred to the screen. An "Illegal Function Call" error will result if the image is too large to fit on the screen.

array
Name of the numeric array which contains the image to be transferred.

action verb
Used to interact the transferred image with the screen. Valid entries are: PSET, PRESET, AND, OR, or XOR. The default is XOR.

The PUT statement transfers the image stored in the array onto the screen.

PSET
Transfers data from the array onto the screen verbatim.

PRESET
This is the same as PSET except that a negative image is produced.

AND
Use AND only when you want to transfer the image to the screen and an image already exists on the screen. Only the points which are in both images will show on the screen.

OR
Use OR to superimpose the image onto an existing image.

XOR
XOR is the default action. It causes the points on the screen to be inverted where a point exists in the array image. You may also use XOR to animate an image. When you PUT an image against a complex background twice, the background remains unchanged. This allows you to move an object around the screen without removing the background.

You may animate an image by following this sequence:

1. Using XOR, PUT the image on the screen.
2. Calculate the new location of the image.
3. Using XOR, PUT the image on the screen a second time at the first location. This action removes the image from the first location.
4. Go back to step 1. Use XOR to PUT the image at the new location.

Movement done this way will leave the background unchanged. You can reduce flicker by minimizing the time between steps 4 and 1 and by making sure that there is enough time delay between steps 1 and 3.

If you are animating more than one image, each image should be processed separately, one step at a time.

If preserving the background is not important, you may animate an image using PSET. However, you must have a rectangle large enough to contain both first and new images. If the area is large enough, the extra area will erase the first image. You may find this method faster than the method using XOR because only one PUT is required, although you must PUT a larger area. In the following example, line 20 sets the dimensions of the screen area to be used. Line 30 draws a filled-in box in color 6, and line 40 reads that box into an array. Lines 50 through 90 PUT the box back on the screen and move it left.

```
10 SCREEN 1
20 DIM M(1000)
30 LINE(0,0)-(30,30),6,BF
40 GET(0,0)-(60,30),M
50 FOR I=579 TO 10 STEP-1
60 PUT(I,200),M,PSET
70 NEXTI
80 GOTO 60
```

AND

	Screen Color								
		0	1	2	3	4	5	6	7
Array Color	0	0	0	0	0	0	0	0	0
	1	0	1	0	1	0	1	0	1
	2	0	0	2	2	0	0	2	2
	3	0	1	2	3	0	1	2	3
	4	0	0	0	0	4	4	4	4
	5	0	1	0	1	4	5	4	5
	6	0	0	2	2	4	4	6	6
	7	0	1	2	3	4	5	6	7

OR

Array Color	Screen Color 0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	1	3	3	5	5	7	7
2	2	3	2	3	6	7	6	7
3	3	3	3	3	7	7	7	7
4	4	5	6	7	4	5	6	7
5	5	5	7	7	5	5	7	7
6	6	7	6	7	6	7	6	7
7	7	7	7	7	7	7	7	7

XOR

Array Color	Screen Color 0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	6	2	3	0	1
7	7	6	5	4	3	2	1	0

4.50 GOSUB...RETURN STATEMENTS

Syntax

GOSUB <line number>
.
.
.
RETURN [<line number>]

Purpose

To branch to, and return from, a subroutine.

Remarks

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

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

Simple RETURN statement(s) in a subroutine cause GW-BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine.

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

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

Example

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
```

```
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
```

will yield

```
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
```

GW-BASIC Compiler

In addition to the simple RETURN statement, the compiler supports RETURN <line number>. This allows a RETURN from a GOSUB statement to the specified line number, rather than a normal return to the statement following the GOSUB statement.

4.51 GOTO STATEMENT

Syntax GOTO <line number>

Purpose To branch unconditionally out of the normal program sequence to a specified line number.

Remarks If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example

```
10 READ R
20 PRINT "R =";R,
30 A=3.14*R∧2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
```

will yield

```
R = 5     AREA = 78.5
R = 7     AREA = 153.86
R = 12    AREA = 452.16
Out of data in 10
```

4.52 HEX$ FUNCTION

Syntax HEX$(X)

Purpose To return a string that represents the hexadecimal value of the decimal argument.

Remarks X is rounded to an integer before HEX$(X) is evaluated.

Example

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

will yield

```
? 32
 32 DECIMAL IS 20 HEXADECIMAL
```

See the OCT$ function, Section 4.86, for details on octal conversion.

4.53 IF...THEN[...ELSE]/IF...GOTO STATEMENTS

Syntax

IF <expression> THEN {<statement(s))> ¦ <line number>}

[ELSE {<statement(s)> ¦ <line number>}]

Syntax

IF <expression> GOTO <line number>

[ELSE {<statement(s)> ¦ <line number>}]

Purpose

To make a decision regarding program flow based on the result returned by an expression.

Remarks

If the result of <expression> is not 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 zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, if executed. Execution continues with the next executable statement. A comma is allowed before THEN.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example,

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

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

will not print "A<>C" where A<>B.

If an IF...THEN statement is followed by a line number in direct mode, an "Undefined line" error results, unless a statement with the specified line number had previously been entered in indirect mode.

Note

When using IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value 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.

Example 1

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

This statement GETs record number I if I is not zero.

Example 2

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

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3

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

This statement causes printed output to go either to the terminal or the line printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the terminal.

GW-BASIC Compiler

The compiler allows indefinite line continuation with the underscore character. Thus, fully nested IF...THEN...ELSE control structures may be set up by using extra-long statements.

4.54 INKEY$ FUNCTION

Syntax INKEY$

Purpose To return either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal.

Remarks No characters will be echoed. All characters are passed through to the program except for Control-C, which terminates the program. (With GW-BASIC Compiler, Control-C is also passed through to the program.)

Example

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

Note Some keys may return a two-byte string, depending on your implementation.

4.55 INP FUNCTION

Syntax INP(I)

Purpose To return the byte read from port I. I must be in the range 0 to 65535.

Remarks INP is the complementary function to the OUT statement.

Example

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

In 8086 assembly language, this is equivalent to:

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

4.56 INPUT STATEMENT

Syntax

INPUT[;] [<“prompt string”>;]<list of variables>

Purpose

To allow input from the keyboard during program execution.

Remarks

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <“prompt string”> is included, the string is printed before the question mark. The required data is then entered at the keyboard.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT “ENTER BIRTHDATE”,B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/linefeed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items or with the wrong type of value (numeric instead of string, etc.) causes the message “?Redo from start” to be printed. No assignment of input

values is made until an acceptable response is given.

Examples

```
10 INPUT X
20 PRINT X "SQUARED IS" X∧2
30 END
will yield
? 5         (The 5 was typed in by the user in
            response to the question mark.)
 5 SQUARED IS 25
```

```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R∧2
40 PRINT "THE AREA OF THE CIRCLE
IS";A
50 PRINT
60 GOTO 20
will yield
WHAT IS THE RADIUS? 7.4 (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

4.57 INPUT# STATEMENT

Syntax

INPUT#<file number>,<variable list>

Purpose

To read data items from a sequential device or file and assign them to program variables.

Remarks

<file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and 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 GW-BASIC 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 will terminate on a comma, carriage return, or linefeed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Example

INPUT#2,A,B,C

4.58 INPUT$ FUNCTION

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

Purpose To return a string of X characters, read from file number Y. If the file number is not specified, the characters will be read from the screen.

Remarks If the keyboard is used for input, no characters will be echoed on the screen. All control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT$ function.

Example 1

```
5 'LIST THE CONTENTS OF A SEQUENTIAL
FILE IN HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2

```
.
.
.
100 PRINT "TYPE P TO PROCEED OR S TO
STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
.
.
.
```

4.59 INSTR FUNCTION

Syntax

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

Purpose

To search for the first occurrence of string Y$ in X$, and returns the position at which the match is found. Optional offset I sets the position for starting the search.

Remarks

I must be in the range 1 to 255. If I is greater than the number of characters in X$ (LEN(X$)), or if X$ is null or Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions, or string literals.

Example

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

will yield

```
 2  6
```

4.60 INT FUNCTION

Syntax INT(X)

Purpose To return the largest integer <=X.

Examples

```
PRINT INT(99.89)
will yield
 99
```

```
PRINT INT(-12.11)
will yield
-13
```

See the CINT and FIX functions, Sections 4.13 and 4.45, respectively, which also return integer values.

4.61 KEY STATEMENT

Syntax

KEY <key number>,<string expression>
KEY LIST
KEY ON
KEY OFF

key number
Specifies the programmable function key number in the range 1 to 20 (see list below).

string expression
Specifies the string expression which will be assigned to the programmable function key.

Purpose

Allows you to assign a string expression to programmable function keys. You may assign a string of up to 15 characters to any one or all of the keys. When you press the key, the string will be input to BASIC.

Remarks

Initially, for GW-BASIC Interpreter, the programmable function keys are assigned the following values:

F1 LOAD"	F11 GOTO⍁
F2 RUN ↵	F12 GOSUB⍁
F3 CONT ↵	F13 IF⍁
F4 SAVE"	F14 THEN⍁
F5 LIST⍁	F15 ELSE⍁
F6 EDIT⍁	F16 CHR$(
F7 TRON ↵	F17 STRING$(
F8 TROFF ↵	F18 LINE⍁(
F9 PRINT⍁	F19 CIRCLE⍁(
F10 PRINT⍁ USING⍁	F20 DRAW⍁

KEY ON
This is the initial setting which causes keys F1 through F7 to be displayed on the 25th line. To display the next seven keys, press CONTROL-T. To display the last six keys, press CONTROL-T again. To start the sequence again, press CONTROL-T.

KEY OFF
Erases the programmable function key display from the 25th line, but it does not disable the function keys.

KEY LIST
Lists all 20 programmable function key values on the screen. All 15 characters of each value are displayed.

KEY <key number>,<string expression>
Assigns the string expression to the specified key. The string expression may be 1 to 15 characters in length. If it is longer than 15 characters, only the first 15 characters are assigned.

If you specify a value for <key number> which is not in the range 1 to 20, an "Illegal Function Call" error occurs. The previous key string assignment is retained.

Assigning a string of length 0 to a programmable function key disables the key. It will remain disabled until another error string expression is assigned to it.

When a programmable function key is assigned, the INKEY$ function returns one character of the string each time it is called. If the programmable function key is disabled, INKEY$ returns a string of length 2. The first character is binary zero, and the second is the key scan code.

Examples

In the following example, the statement in line 10 assigns the string 'MENU'<carriage return> to key F1. This assignment might be used in a program to select a menu display when entered by the user. Line 20 disables the key.

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

The following routine initializes the first 5 programmable function keys:

```
10 KEY OFF
20 DATA KEY1,KEY2,KEY3,KEY4,KEY5
30 FOR I=1 to 5:READ FUNCTIONKEYS$(I)
40 KEYI,FUNCTIONKEYS$(I)
50 NEXT I
60 KEY ON
```

GW-BASIC Compiler

With the compiler, programmable function key string values are not preserved across chains.

4.62 KEY(N) STATEMENT

Syntax

KEY(n) ON
KEY(n) OFF
KEY(n) STOP

where (n) is the number of a programmable function key or cursor direction key. (See "KEY Statement," Section 4.61, for information on assigning programmable function key values to function keys.) The cursor direction keys are numbered sequentially after the function keys in the following order: up, left, right, down.

Purpose

To enable or disable event trapping of programmable function key or cursor direction key activity for the specified function key.

Remarks

Note that the KEY statement described in Section 4.61 assigns programmable function key and cursor direction values to function keys and displays the values. Do not confuse KEY ON and KEY OFF, which display and erase these values, with the event trapping statements described in this section.

The KEY(n) ON statement enables programmable function key or cursor direction key event trapping by an ON KEY statement (see "ON KEY Statement," Section 4.90). While trapping is enabled, and if a non-zero line number is specified in the ON KEY statement, GW-BASIC checks between every statement to see if a programmable function key or cursor direction key has been used. If it has, the ON KEY statement is executed.

KEY(n) OFF disables the event trap. If an event takes place, it is not remembered.

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

Note

For additional information on key event trapping, see "Event Trapping," Section 1.6, and "ON KEY Statement," Section 4.90.

Example

```
10 KEY 4,SCREEN 0,0 ' assigns programmable
function key 4
20 KEY(4) ON 'enables event trapping
 .
70 ON KEY(4) GOSUB 200
 .
 .
 .
key 4 pressed
 .
 .
 .
200 'Subroutine for screen
```

GW-BASIC Compiler

See compiler note under "ON KEY Statement," Section 4.90.

4.63 KILL STATEMENT

Syntax

KILL <filespec>

Purpose

To delete a file from disk.

Remarks

If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files, and sequential data files. The filespec may contain question marks (?) or asterisks (*) used as wildcards. A question mark will match any single character in the filename or extension. An asterisk will match one or more characters starting at its position.

WARNING: Be extremely careful when using wildcards with this command.

Examples

200 KILL "DATA1?.DAT"

The position taken by the question mark can contain any valid filename character.

210 KILL "DATA1.*"

Kills all files named DATA1.

4.64 LCOPY STATEMENT

Syntax

LCOPY

Purpose

Prints the screen memory (1, 2, 4) established with the GW-BASIC configuration routine.(See "Configuring for GW-BASIC" in your User's Guide.)

Remarks

You may use LCOPY to print both graphics images and text if the text is also in graphics mode.

With a monochrome model LCOPY prints the entire graphics image (and text in graphics mode).

The GW-BASIC Interpreter disk contains an object module which you may use for program control of color graphics printing. The module is called DUMPCL (dump color) and allows you to specify within a program a color memory (1,2,4) to be printed.

Before you can use DUMPCL, it must be established as a separate file on your GW-BASIC disk. This is accomplished by using MS-LINK (see MS-DOS User's Guide) and DEBUG, which allows you to read the address of the DUMPCL module. These are on your MS-DOS disk. If you have a single flexible disk drive, follow the sequence below to establish DUMPCL as a file. If you have 2 flexible disk drives, note the instructions in parentheses in each step.

1. Insert the MS-DOS disk. (For 2 disk drives, insert the MS-DOS disk and the GW-BASIC disk.)
2. The system displays A>. Enter LINK. (For 2 disk drives, move operations to drive B. Enter B:, and when the system displays B>, enter A:LINK.)
3. The system displays:

Microsoft Object Linker V2.00
(c) Copyright 1982 by Microsoft Inc.

Object Modules [.OBJ]:

Insert the GW-BASIC disk. Enter DUMPCL. (For 2 disk drives, simply enter DUMPCL.)

4. The system displays:

   Run File [DUMPCL.EXE]:

   Enter /H to specify the highest address in memory. (For 2 disk drives, enter B: /H.)

5. The system displays:

   List File [NUL.MAP]:

   Press RETURN.

6. The system displays:

   Libraries [.LIB]:

   Press RETURN.

7. The system displays:

   Warning: No STACK segment

   There was 1 error detected.

   A>

   The warning and error detected do not affect the procedure. Insert the MS-DOS disk and enter DEBUG. (For 2 disk drives, the prompt will be B>. Enter A:DEBUG DUMPCL.EXE. The DUMPCL.EXE file is loaded directly into DEBUG.)

8. The system displays a dash (-). Insert the GW-BASIC disk and enter NDUMPCL.EXE. This specifies the new DUMPCL.EXE file to DEBUG. (For 2 disk drives, enter R to display all registers. Then go to step 11.)
9. The system displays a dash (-). Enter L to load DUMPCL.EXE into DEBUG.

10. The system displays a dash (-). Enter R to display all registers.
11. The system displays the following:

```
AX=FFFF BX-0000   CX-0026  DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=0AA0 ES-0AA0   SS=1FE0 CS=1FE0 IP=0000 NV UP DI PL NZ NA PO NC
1FE0:0000 55               PUSH   BP
```

Note the address CS=1FE0. CS represents the address of the DUMPCL module. The address given here (1FE0) is only an example. You should note the address for CS on your machine. Write it down for later use.

12. The system displays a dash (-). Enter NGWBASIC.EXE. This specifies the GWBASIC.EXE file.
13. The system displays a dash (-). Enter L to load GWBASIC.EXE to memory.
14. The system displays a dash (-). Enter G to go to GWBASIC.EXE.

You are now in GW-BASIC.

15. Enter the following lines:

```
DEF SEG=&H1FE0
BSAVE "DUMPCL.COM",&H0,&H30
```

Note that the DEF SEG value entered is the address of the DUMPCL module. You should insert the value for CS that was displayed on your machine (Step 11). DUMPCL.COM in the second line is the new file name to be saved. The last two entries of the second line are the offset address from the address given in the DEF SEG statement and the length in bytes of the file to be saved (the DUMPCL file always has the length given here). Refer to the BSAVE statement in the MS-DOS Extension, Section 2.

The file is now saved on your disk. Exit GW-BASIC by entering SYSTEM. Exit DEBUG by entering Q. You may now load GW-BASIC.

Now that you have your DUMPCL.COM file on disk, you may use it for program control of color graphics printing. To use your file, you must include the following BASIC statements in your program:

```
10 DEF SEG=&H1FE0
20 BLOAD"DUMPCL.COM",&H0
30 A%=1
40 CALL &H0(A%)
50 LCOPY
```

In line 10 you will enter the address of the DUMPCL module which was displayed with step 11. In line 30, specify the color memory your want printed. Enter 1 for blue foreground, 2 for green, and 4 for red.

Example

For an example of selecting screen images for printing, see "Color Selection" in Chapter 1.

GW-BASIC Compiler

The compiler uses GWCONF only to set-up memory print; consequently, the program control of color graphics printing mentioned above does not apply. For monochrome machines, however, you must select green foreground color dump before executing your program.

4.65 LEFT$ FUNCTION

Syntax LEFT$(X$,I)

Purpose To return a string comprising the leftmost I characters of X$.

Remarks I must be in the range 0 to 255. If I is greater than the number of characters in X$ (LEN(X$)), the entire string (X$) will be returned. If I = 0, the null string (length zero) is returned.

Example

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

will yield

BAS

Also see the MID$ and RIGHT$ functions, Sections 4.82 and 4.116, respectively.

4.66 LEN FUNCTION

Syntax LEN(X$)

Purpose To return the number of characters in X$. Nonprinting characters and blanks are counted.

Example

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

will yield

```
 16
```

4.67 LET STATEMENT

Syntax [LET]<variable> = <expression>

Purpose To assign the value of an expression to a variable.

Remarks Notice that the word LET is optional; i.e., the equal sign is sufficient for assigning an expression to a variable name.

Example

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

or

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

4.68 LINE STATEMENT

Syntax

LINE [(x1,y1)]-(x2,y2)[,[color][b[f]]

Purpose

Draws a line, box, or filled-in box on the screen.

(x1,y1),(x2,y2)
Specifies the coordinates in either absolute or offset form (see X and Y coordinates in Chapter 1). If (x1,y1) point coordinates are not specified, the beginning point of the line is the last point specified by (x2,y2) in a previous statement.

color
Specifies color of line, box, or filled-in box (0-7, see Color statement, Section 4.18). If not specified, color is foreground color.

b or bf
Specifies box or filled-in box. The b tells BASIC to draw a rectangle with the points (x1,y1) and (x2,y2) as opposite corners. This avoids having to give four LINE commands which perform the same function:

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

The bf tells BASIC to draw the same rectangle as b and also to fill in the interior points in the same color as b.

Remarks

The offset coordinate form can be used wherever a coordinate is used. Note that all of the graphics statements and functions update the most recent point used. If the offset form is used with the second coordinate, the coordinate is offset from the first coordinate in the statement. For example, the following draws a line from (60,40) to 70,50):

```
LINE (60,40)-STEP(10,10)
```

If you specify a coordinate which is out of range, the coordinate is given the closest valid value. In other words, negative values become 0, y values greater than 399 become 399, and x values greater than 639 become 639.

Examples

Draw a line from the last point specified to the point (x2,y2):

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

Include a starting point for a line diagonally down the screen:

```
LINE (0,0)-(639,399)
```

Draw a line across the screen:

```
LINE (0,200)-(639,200)
```

Draw a line in color number 2:

```
LINE (10,0)-(20,20),2
```

Draw a box in foreground color:

```
LINE (0,0)-(100,100),,b
```

Draw a box and fill it in with color number 2:

```
LINE (0,0)-(200,200),2,bf
```

Draw lines continuously using random colors:

```
10 SCREEN 1
20 CLS
30 LINE -(rnd*639,rnd*399),rnd*7
40 GO TO 20
```

Draw alternating pattern - line on, line off:

```
10 SCREEN 1
20 FOR X=0 TO 639
30 LINE (X,0)-(X,399),X AND 1
40 NEXT
```

Draw random filled boxes in random colors:

```
10 SCREEN 1
20 CLS
30 LINE -(rnd*639,rnd*399),rnd*7,bf
40 GO TO 20
```

4.69 LINE INPUT STATEMENT

Syntax

LINE INPUT[;] [<"prompt string">;] <string variable>

Purpose

To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks

<"prompt string"> is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of <"prompt string">. All input from the end of <"prompt string"> to the carriage return is assigned to <string variable>. However, if a linefeed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the linefeed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/linefeed sequence at the terminal.

A LINE INPUT statement may be aborted by typing Control-C. GW-BASIC will return to command level. If you are using the interpreter, typing CONT resumes execution at the LINE INPUT.

Example

See "LINE INPUT# Statement," Section 4.70.

4.70 LINE INPUT# STATEMENT

Syntax

LINE INPUT#<file number>,<string variable>

Purpose

To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks

<file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/linefeed sequence. The next LINE INPUT# reads all characters up to the next carriage return. (If a linefeed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a GW-BASIC program saved in ASCII format is being read as data by another program. (See "SAVE Command," Section 4.119.)

Example

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER
INFORMATION? ";C$
30 PRINT #$1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
```

will yield

```
CUSTOMER INFORMATION? LINDA
JONES    234,4    MEMPHIS
LINDA JONES    234,4    MEMPHIS
```

4.71 LIST STATEMENT

Syntax

LIST [[<line number>[- [<line number>]]] [,<dev<]]

line number
Specifies the line number in the range 0 to 65529.

dev
String expression for one of the following devices:

"SCRN:"	Screen
"LPT1:"	Line printer

Purpose

Allows a program to be listed to the screen or line printer.

Remarks

If you omit the device argument, the device is the screen.

If you omit the line argument, the software lists the entire program.

You may interrupt any listings to either the screen or the printer by pressing CONTROL-S.

If you use the dash (-) in the line argument, the following three options are available:

If you specify only the first line number, the software lists that line and all higher numbered lines.

If you specify only the second line number, the software lists all lines from the beginning of the program through the specified line.

If you specify both line numbers, the software lists the inclusive range.

LIST ,"LPT1:" is the same as LLIST in MS-BASIC. If "LPT1:" is specified, the last valid width command entered for the line printer will be used.

LIST ,"SCRN:" is the same as LIST in MS-BASIC.

Examples

List program to the line printer:

LIST ,"LPT1:"

List lines 10 through 20 to the screen:

LIST 10-20

List lines 10 through the last line to the screen:

LIST 10- ,"SCRN:"

List the first line through line 200 to the line printer:

LIST -200,"LPT1:"

List lines 35 through 65 to the screen:

LIST 35-65,"SCRN:"

GW-BASIC Compiler

The LIST command is not supported by the GW-BASIC Compiler.

4.72 LOAD COMMAND

Syntax

LOAD <filespec>[,R]

Purpose

To load a file from disk into memory.

Remarks

The <filespec> must include the filename that was used when the file was saved. (Your operating system may append a default filename extension if one was not supplied in the SAVE command.)

The R option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the R option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the R option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example

LOAD "STRTRK",R

Loads and runs the program STRTRK.

LOAD "B:MYPROG"

Loads the program MYPROG from the disk in drive B, but does not run the program.

GW-BASIC Compiler

The LOAD command is not supported by the GW-BASIC Compiler.

4.73 LOC FUNCTION

Syntax

LOC(<file number>)

where <file number> is the number under which the file was opened.

Purpose

With random disk files, LOC returns the record number of the last record read or written.

With sequential files, LOC returns the number of records read from, or written to, the file since it was opened.

Remarks

When a file is opened for sequential input, GW-BASIC 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(X) is used to determine if there are any characters in the input queue waiting to be read. If there are more than 255 characters in the queue, LOC(X) returns 255. Since interpreter strings are limited to 255 characters, this practical limit alleviates the need for an interpreter user to test for string size before reading data into it.

If fewer than 255 characters remain in the queue, the value returned by LOC(X) depends on whether the device was opened in ASCII or binary mode. In either mode, LOC will return the number of characters that can be read from the device. However, in ASCII mode, the low level routines stop queueing characters as soon as end-of-file is received. The end-of-file itself is not queued and cannot be read. An attempt to read the end-of-file will result in an "Input past end" error.

Example

```
200 IF LOC(1)>50 THEN STOP
```

4.74 LOCATE STATEMENT

Syntax

LOCATE [row] [,[col] [,[cursor]]]

row
Specifies screen line number. Optional. Specify a numeric expression which must result in an unsigned integer in the range 1 to 24.

col
Specifies screen column number. Optional. Specify a numeric expression which must result in an unsigned integer in the range 1 to 80.

cursor
A boolean value indicating whether the cursor is visible. Optional. Specify 0 for off, non-zero for on.

Purpose

Moves cursor to specified position on the active screen. Subsequent PRINT statements begin placing characters at this location. Optional cursor argument turns the blinking cursor on and off.

Remarks

If you enter values outside the ranges for row and column, an "Illegal Function Call" error occurs. Previous values are retained.

You may omit any argument. Omitted arguments assume the previous value.

You cannot select cursor blink frequency.

The 25th line is reserved for the programmable function key display. As a recommendation, do not write over the line, even if the display is off.

Example

In the following example, the statement in line 10 moves the cursor to the home position in the upper left hand corner. The statement in line 20 makes the blinking cursor visible without changing its position.

```
10 LOCATE 1,1
20 LOCATE ,,1
```

4.75 LOF FUNCTION

Syntax LOC(<file number>)

Purpose To return the length of the file in bytes.

Example
```
110 IF REC*RECSIZ>LOF(1)
THEN PRINT "INVALID ENTRY"
```

In this example, the variables REC and RECSIZ contain the record number and record length, respectively. The calculation determines whether the specified record is beyond the end-of-file.

4.76 LOG FUNCTION

Syntax	LOG(X)
Purpose	To return the natural logarithm of X. X must be greater than zero.
Example	PRINT LOG(45/7) will yield 1.860752

4.77 LPOS FUNCTION

Syntax LPOS(X)

where X is the number assigned to the line printer.

Purpose To return the current position of the line printer's print head within the line printer buffer.

Remarks LPOS does not necessarily give the physical position of the print head.

Example 100 IF LPOS(X)>60 THEN LPRINT CHR$(13)

4.78 LPRINT AND LPRINT USING STATEMENTS

Syntax

LPRINT [<list of expressions>]

LPRINT USING <string exp>;<list of expressions>

Purpose

To print data at the line printer.

Remarks

Same as PRINT and PRINT USING, except output goes to the line printer. See Sections 4.103 and 4.104, respectively.

LPRINT assumes a 132-character-wide printer. However, the width may vary according to your implementation.

Examples

See Sections 4.103 and 4.104.

4.79 LSET AND RSET STATEMENTS

Syntax

LSET <string variable>=<string expression>
RSET <string variable>=<string expression>

Purpose

To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks

If <string expression> requires fewer bytes than were fielded to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See "MKI$, MKS$, MKD$," Section 4.83.

Examples

```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC($)
```

Note

LSET or RSET may also be used with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

4.80 MERGE COMMAND

Syntax

MERGE <filespec>

Purpose

To merge a specified disk file into the program currently in memory.

Remarks

The <filespec> must include the filename used when the file was saved. (Your operating system may append a default filename extension if one was not supplied in the SAVE command.) The file must have been saved in ASCII format. If it was not, a "Bad file mode" error occurs.

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 will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

GW-BASIC always returns to command level after executing a MERGE command.

Example

MERGE "NUMBERS"

Inserts, by sequential line number, all lines in the program NUMBRS into the program currently in memory.

GW-BASIC Compiler

The MERGE command is not supported by the GW-BASIC Compiler.

4.81 MID$ STATEMENT

Syntax

MID$(<string expl>,n[,m])=<string exp2>

where n and m are integer expressions and <string expl> and <string exp2> are string expressions.

Purpose

To replace a portion of one string with another string.

Remarks

The characters in <string expl>, beginning at position n, are replaced by the characters in <string exp2>. The optional "m" refers to the number of characters from <string exp2> that will be used in the replacement. If "m" is omitted, all of <string exp2> is used. However, regardless of whether "m" is omitted or included, the replacement of characters never goes beyond the original length of <string expl>.

Example

```
10 A$="KANSAS CITY, NO"
20 MID$(A$,14)="KS"
30 PRINT A$
will yield
KANSAS CITY, KS
```

MID$ is also a function that returns a substring of a given string. See Section 4.82.

4.82 MID$ FUNCTION

Syntax MID$(X$,n[,m])

Purpose To return a string of length m characters from X$, beginning with the nth character.

Remarks n and m must be in the range 1 to 255. If m is omitted or if there are fewer than m characters to the right of the nth character, all rightmost characters beginning with the nth character are returned. 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)
```

will yield

```
GOOD EVENING
```

Also see the LEFT$ and RIGHT$ functions, Sections 4.65 and 4.116, respectively.

4.83 MKI$, MKS$, MKD$ FUNCTIONS

Syntax

MKI$ (<integer expression>)
MKS$ (<single precision expression>)
MKD$ (<double precision expression>)

Purpose

To convert numeric values to string values.

Remarks

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.

Example

```
90 AMT=(K+T)
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$=MKS$(AMT)
120 LSET N$=A$
130 PUT #1
 .
 .
```

See also "CVI, CVS, CVD Functions," Section 4.25.

4.84 NAME STATEMENT

Syntax NAME <old filename> AS <new filename>

Purpose To change the name of a disk file.

Remarks <old filename> must exist and <new filename> must not exist; otherwise, an error will result.

A file may not be renamed with a new drive designation. If this is attempted, a "Rename across disks" error will be generated. After a NAME command, the file exists on the same disk, in the same area of disks space, with the new name.

Example NAME "ACCTS" AS "LEDGER"

In this example, the file that was formerly named ACCTS will now be named LEDGER.

4.85 NEW COMMAND

Syntax NEW

Purpose To delete the program currently in memory and clear all variables.

Remarks NEW is entered in direct mode to clear memory before entering a new program. GW-BASIC always returns to command level after a NEW is executed.

NEW closes all files and turns tracing off.

Example NEW

GW-BASIC Compiler The NEW command is not supported by the GW-BASIC Compiler.

4.86 OCT$ FUNCTION

Syntax OCT$(X)

Purpose To return a string that represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example PRINT OCT$(24)
will yield
 30

See the HEX$ function, Section 4.52, for details on hexadecimal conversion.

4.87 ON COM(n) STATEMENT

Syntax

ON COM(n) GOSUB <line>

n
Communications channel number (1 or 2).

line
Line number of the beginning of the trap routine. A line number of 0 disables trapping for the specified channel.

Function

Allows the software to trap a line number when information comes into the communications buffer.

Remarks

The following statements control the activation or deactivation of the trapping routine:

COM(n) ON
Must be performed to activate the ON COM(n) statement. If you specify a non-zero line in the ON COM (n) statement, every time the program starts a new statement, the software checks to see if any characters have come into the specified channel. If there are no characters, the software performs a GOSUB to the specified line.

COM(n) OFF
If performed, no trapping takes place for the channel. Even if there are communications, the characters received by the channel are not saved in memory.

COM(n) STOP
If performed, no trapping takes place for the channel. However, any characters received by the channel are saved in memory so that an immediate trap takes place when COM(n) ON is performed.

When a trap occurs, the trap automatically causes a COM(n) STOP on that routine so that recurring traps can never take place. The

RETURN from the trap routine automatically performs COM(n) ON unless an explicit COM(n) OFF has been performed within the trap routine.

Trapping never takes place unless the software is executing a program.

When an error trap takes place, all trapping is automatically disabled.

Typically, the communications trap routine reads an entire message from the communications channel before returning back. It is not recommended to use the communications trap for single character messages because at high baud rates the overhead of trapping and reading for each individual character may cause the interrupt buffer for communications to overflow.

RETURN <line>
This form of RETURN is optional. Use it to go back to the software program at a fixed line number. This action eliminates the GOSUB entry which the trap created. Use RETURN <line> with care! Any other GOSUB, WHILE, or FOR which was active at the time of the trap will remain active. If a trap returns from a subroutine, any attempt to continue loops outside the subroutine will result in a "NEXT without FOR" error.

GW-BASIC Compiler

With the compiler, the /V or /W switch must be given in the compiler command line if a program contains an ON COM statement. These switches allow the compiler to function correctly when event trapping routines are included in a program. See your GW-BASIC Compiler User's Guide for an explanation of these switches.

4.88 ON ERROR GOTO STATEMENT

Syntax

ON ERROR GOTO <line number>

Purpose

To enable error handling and specify the first line of the error handling routine.

Remarks

Once error handling has been enabled, all errors detected, including direct mode errors (e.g., syntax errors), will cause a jump to the specified error handling routine. If <line number> does not exist, an "Undefined line" error results.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error handling routine causes GW-BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note

If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

Example

```
10 ON ERROR GOTO 1000
```

GW-BASIC Compiler

With the compiler, the /E compilation switch must be given in the compiler command line if a program contains ON ERROR GOTO and RESUME <line number> statements. If the RESUME, RESUME NEXT, or RESUME 0 form is used, thee /X switch must be specified instead.

The purpose of these switches is to allow the compiler to function correctly when error handling routines are included in a program. See the *NCR GW-BASIC Compiler User's Guide* for an explanation of these switches.

4.89 ON...GOSUB AND ON...GOTO STATEMENTS

Syntax

ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Purpose

To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks

The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a noninteger, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), Microsoft GW-BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example

```
100 ON L-1 GOTO 150,300,320,390
```

GW-BASIC Compiler

The compiler does not check the value of expression, except to make sure that it does not exceed the number of items in the <list of line numbers>. If that is the case, erroneous results will be produced.

4.90 ON KEY(N) STATEMENT

Syntax

ON KEY(n) GOSUB <line>

n
Specifies a function key numbered 1 through 24 as follows:

1-20	Softkeys F1 through F20
21	Cursor up
22	Cursor left
23	Cursor right
24	Cursor down

line
Specifies the line number where BASIC will begin the trapping routine for the specified key. A line number of 0 disables trapping of the key.

Purpose

Allows the software to trap a line number when you press the specified function key or cursor key.

Remarks

The following statements control the activation or deactivation of the trapping routine:

KEY(n) ON
Must be performed to activate the ON KEY(n) statement. If you specify a non-zero line for the trap with ON KEY(n), every time the program starts a new statement, the software checks to see if the specified key was pressed. If you pressed the key, the software performs a GOSUB to the specified line.

KEY(n) OFF
If performed, no trapping takes place for the specified key. Even if you press the key, the trap routine is not remembered.

KEY(n) STOP
If performed, no trapping takes place for the specified key. However, if you press the specified key, an immediate trap takes place when KEY(n) ON is performed.

When a trap occurs, the trap automatically causes a KEY(n) STOP on that routine so that recurring traps can never take place. The RETURN from the trap routine automatically performs a KEY(n) ON unless an explicit KEY(n) OFF has been performed within the trap routine.

Trapping never takes place unless the software is executing a program.

When an error trap takes place, all trapping is automatically disabled.

No type of trapping is activated when the software is in direct mode. In particular, function keys resume their standard expansion meaning during input.

A key that causes a trap cannot be tested with the INPUT or INKEY$ statements, so the trap routine for each key must be different if you want a different function.

RETURN <line>
This form of RETURN is optional. Use RETURN <line> to go back to the software program at a fixed line number. This action eliminates the GOSUB entry which the trap created. Use RETURN <line> with care! Any other GOSUB, WHILE, or FOR which was active at the time of the trap will remain active. If a trap returns from a subroutine, any attempt to continue loops outside the subroutine will result in a "NEXT without FOR" error.

GW-BASIC Compiler

With the compiler, the /V or /W switch must be given in the compiler command line if a program contains an ON KEY statement. These switches allow the compiler to function correctly when event trapping routines are included in a program. See the GW-BASIC Compiler User's Guide for an explanation of these switches.

4.91 ON STRIG STATEMENT

Syntax

ON STRIG(n) GOSUB <line number>

where (n) is the number of the joystick trigger.

where <line number> is the number of the first line of a subroutine that is to be performed when the joystick trigger is pressed.

Purpose

To specify the first line number of a subroutine to be performed when the joystick trigger is pressed.

Remarks

A <line number> of zero disables the event trap.

The ON STRIG statement will only be executed if a STRIG ON statement has been executed (see "STRIG Statement/Function," Section 4.131) to enable event trapping. If event trapping is enabled, and if the < line number> in the ON STRIG statement is not zero, GW-BASIC checks between statements to see if the joystick trigger has been pressed. If it has, a GOSUB will be performed to the specified line.

If a STRIG OFF statement has been executed (see "STRIG Statement," Section 4.131), the GOSUB is not performed and is not remembered.

If a STRIG STOP statement has been executed (see "STRIG Statement," Section 4.131), the GOSUB is not performed, but will be performed as soon as a STRIG ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic STRIG STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a STRIG ON statement unless an explicit STRIG OFF was performed inside the subroutine.

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 GW-BASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

GW-BASIC Compiler

With the compiler, the /V and /W switch must be given in the compiler command line if a program contains an ON STRIG statement. These switches allow the compiler to function correctly when event trapping routines are included in a program. See your GW-BASIC Compiler User's Guide for an explanation of these switches.

4.92 OPEN STATEMENT

Syntaxes

OPEN <mode>,[#]<file number>,<filespec> [,<record length>]

OPEN <filespec>[FOR <mode>] AS [#]<file number> [LEN=<record length>]

<mode> is a string expression whose first character is one of the following:

O Specifies sequential output mode.

I Specifies sequential input mode.

R Specifies random input/output mode.

A Specifies sequential output mode and sets the file pointer at the end of file and the record number as the last record of the file. A PRINT# or WRITE# statement will then extend (append) the file.

If <mode> is omitted, the default random access mode is assumed.

<file number> is an integer expression whose value is between 1 and 15. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filespec> is a string expression containing a name that conforms to your operating system's rules for disk filenames.

<record length> is an integer expression that, if included, sets the record length for random files. Do not use this option with sequential files.

With the interpreter, the <record length> cannot exceed the maximum set with /S: at start-up. If the <record length> option is not used, the default length is 128 bytes.

Purpose

To allow I/O to a file or device.

Remarks

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.

Note

A file can be opened for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Examples

```
10 OPEN "I",2,"INVEN"

10 OPEN "MAILING.DAT" FOR APPEND AS
   1
```

OPEN "O",2,"SCRN:"

4.93 OPEN COM STATEMENT

Syntax

OPEN "<dev>:[<speed>],[<parity>],[<data>],[<stop>][,RS][,CS[<n>]][,DS[<n>]][,CD[<n>]] [,LF]" AS [#]<file number>

Purpose

Opens a communications file. Allocates a buffer for I/O in the same manner as OPEN for disk files. Supports RS-232 asynchronous communication with other computers and peripherals.

Remarks

dev
Specifies one of the following communications devices: COM1 or COM2.

speed
An integer constant which specifies the transmit or receive baud rate. Valid speeds are: 50, 75, 100, 134, 150, 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600, 19200. Baud rate 134 includes 134.5. Default is 300 bps.

parity
A one-character constant which specifies the parity for transmit and receive, as follows:

S SPACE: parity bit is always transmitted and received as a space (0 bit).

O ODD: odd transmit and receive parity checking.

M MARK: parity bit is always transmitted and received as a mark (1 bit).

E EVEN: even transmit and receive parity checking.

N NONE: no transmit or receive parity checking.

The default for parity is even (E).

data
An integer constant which indicates the number of transmit or receive data bits. Valid values are: 4, 5, 6, 7, and 8. The default is 7. If you specify 4, you must also specify mark (M) or space (S) parity. If you do not specify M or S, a "Bad File Name" error occurs. If you specify 8 bits, you must specify N (none) parity.

stop
An integer constant which indicates the number of stop bits. Valid values are 1 and 2. The default stop bit for 50,75 and 110 bps is 2. The default for all others is 1. If you specify 4 or 5 for <data>, a 2 entered for <stop> will mean 1 1/2 stop bits.

RS
Suppresses Request To Send (RTS) line signal. If you enter RS, the RTS line is not turned on when an OPEN COM statement is run.

CS<n>
Controls Clear To Send (CTS) line signal. If you enter CS, the system waits for the line signal without returning an error. If you enter CSn, n specifies the amount of time to wait before the system returns a "Device Timeout" error. Setting n equal to zero is the same as entering CS. If you omit the option, the default is 1 second.

DS<n>
Controls Data Set Ready (DSR) line signal. If you enter DS, the system waits for the line signal without returning an error. If you enter DSn, n specifies the amount of time to wait before the system returns a "Device Timeout" error. Setting n equal to zero is the same as entering DS. If you omit the option, the default is 1 second.

CD<n>
Controls Carrier Detect (CD) line signal, also known as Received Line Signal Detect (RLSD). If you enter CD, the line signal is not checked. If you enter CDn, n specifies the amount of time to wait

before the system returns a "Device Timeout" error. If you set n equal to zero or you omit the option, the line signal is not checked.

n
Specifies the number of milliseconds the system will wait before returning a "Device Timeout" error. n may range from 0 to 65535.

LF
Sends a line feed following each carriage return. Specify LF when using communication files to print to a serial line printer. Note that INPUT# and LINE INPUT#, when used to read from a communications file which was opened with the LF option, ignore the line feed and stop when they detect a carriage return.

file number
Specifies an integer expression which returns a valid file number. The number is associated with the file for as long as it is open and is used by other communications I/O statements to refer to the file.

Any coding errors within the string expression from <speed> through LF result in a "Bad File Name" error. An indication of which parameter is in error is not given.

If the communications adapter is not correctly in place, a "Device Timeout" error occurs when DSR is not detected. Refer to the hardware documentation for proper cabling instructions.

See the Communications chapter for information on communications I/O. Error messages for communications are included in the Error Messages appendix.

Example

In the following example, file number 1 is opened for communication with all defaults: 300 bps, even parity, and 7 data bits with 1 stop bit.

```
10 OPEN "COM1:" AS #1
```

The following opens file number 2 for communication at 2400 bps. Defaults are: even parity, 7 data bits, and 1 stop bit.

```
10 OPEN "COM1:2400" AS #2
```

The following opens file number 1 for asynchronous I/O at 1200 bps. No parity is produced or checked. 8-bit bytes will be sent and received. The stop bit is defaulted to 1.

```
10 OPEN "COM2:1200,N,8" AS #1
```

OPTION

4.94 ~~OPEN~~ BASE STATEMENT

Syntax

OPTION BASE n

where n is 1 or 0

Purpose

To declare the minimum value for array subscripts.

Remarks

The default base is 0. If the statement

OPTION BASE 1

is executed, the lowest value an array subscript may have is 1.

The OPTION BASE statement must be coded before you define or use an arrays.

Example

```
10 OPTION BASE 1
```

GW-BASIC Compiler

The compiler does not "execute" an OPTION BASE statement, as it does a PRINT statement, for example. An OPTION BASE statement takes effect as soon as it is encountered in the program during compilation. This option base then remains in effect until the end of the program or until another OPTION BASE statement is encountered.

4.95 OUT STATEMENT

Syntax

OUT I,J

where I is the port number. It must be an integer expression in the range 0 to 65535.

J is the data to be transmitted. It must be an integer expression in the range 0 to 255.

Purpose

To send a byte to a machine output port.

Example

```
100 OUT 12345,255
```

In 8086 assembly language, this is equivalent to:

```
MOV DX,12345
MOV AL,255
OUT DX,AL
```

4.96 PAINT STATEMENT

Syntax

Fills in an area on the screen with the specified color.

x,y
Specify the coordinates of the point where painting begins. The point, which may be given in absolute or offset form (see Chapter 1 for an explanation of x and y coordinates), may be inside or outside a figure, but not a boundary.

paint color
Specifies the color to be painted (the "fill" color). Enter a value from 0-7 (see Color statement, section 4.18).

boundary color
Specifies the boundary color of the figure. Enter a value from 0-7.

Remarks

The starting point may be inside or outside a figure. If the point is outside, the screen is painted with the color, except for the area inside the boundary..

If you do not specify a paint color, the currently active foreground color is used; if you do not specify a boundary color, the entire screen is painted.

For monochrome screens, you may specify only black or green. If you specify any other color, green is automatically used.

Example

The following example draws a circle and then paints it first with yellow and then with black. With the GOTO statement, the sequence is repeated, causing an animated effect.

```
10 COLOR 7,0
20 CIRCLE (300,200),10,1
40 PAINT (300,191),6,1
50 PAINT (300,209),0,0
60 GOTO 20
```

Note that the boundary color on the second statement is black, which covers the previous blue boundary and "erases" the circle.

4.97 PEEK FUNCTION

Syntax PEEK(I)

Purpose To return the byte read from the indicated memory location (I).

Remarks The returned value is an integer in the range 0 to 255. I must be in the range -32768 to 65535. I is the offset from the current segment, which was defined by the last DEF SEG statement (see Section 4.31). For the interpretation of a negative value of I, see "VARPTR Function," Section 4.142.

PEEK is the complementary function of the POKE statement.

Example A=PEEK(&H5A00)

4.98 PLAY STATEMENT

Syntax PLAY <string expression>

Remarks With the Play statement, you can generate or create a tune by defining its characteristics in the string expression. The expression may consist of any of the following commands, which you may specify in any order unless stated otherwise in the description.

A-G [#,+,-] - Music Scale
Plays the specified notes, A-G. A# or + after a note specifies a sharp (a half step higher in pitch); a - after a note specifies a flat (a half step lower in pitch).

L <n> - Length
Sets the length of the note (or notes), where n may be from 1 to 64. As examples, L1 specifies a hole note, L2 specifies a half note...and L64 specifies a sixty-fourth note. You may specify the length before a group of notes or after a single note to change only its length. In the latter case, for example, A16 is the same definition as L16A.

MB - Music Foreground
Sets music or any sound to run in the foreground. Each subsequent note or sound is not started until the previous note or sound is finished. MF is the initial default value.

MN - Music Normal
Plays each note 7/8ths. of the time specified in L (length).

ML - Music Legato
Plays each note the full length (as specified in L).

MS - Music Staccato
Plays each note 3/4ths. of the time specified in L (length).

N <n> - Note
Plays the note specified by n. n may range from 22 to 63. (See table of notes under the sound statement.) n may equal 0 to specify a pause. Using this command provides an alternative way to specify the note other than by name (A-G) and octave.

O <n> - Octave
Sets the octave, where n may range from 1 to 5.

P <n> - Pause
Sets the length of the pause, where n may range from 1 to 64. The n value is the same as the n value in the Length command; for example, P1 causes a pause the length of a whole note, P2 causes a pause the length of a half note, and so on.

T <n> - Tempo
Sets the number of quarter notes (n) that can be played in a minute. n may range from 32 to 255; the default value is 120.

. - dot or period
Used after a note, plays the note as a dotted note; that is, its length is multiplied by 3/2. More than one dot may be used after the note, in which case its length is adjusted accordingly. As examples, A.. plays 9/4 as long as L specifies, A... plays 27/8 as long, etc. Dots may also be used after a pause (P) to scale the pause length in the same way.

X variable;
Executes specified string. (Not available with GW-BASIC Compiler.)

In all commands, the n value can be a constant or = variable; where variable is the name of a variable. The semicolon (;) is required when you use a variable in this way, and when you use the X command; otherwise, a semicolon is optional between commands, except it is not allowed after MF, MB, MN, ML, or MS. Blanks in a string are ignored.

You can also specify variables in the form VARPTR$ (variable), instead of = variable;. This method is useful in programs that will later be compiled.

Examples

The following two examples are equivalent; the first is used with the interpreter, the second with the compiler.

PLAY "XA#;"

PLAY "X"+VARPTR$(A$)

You can use X to store a "subtune" in one string and call it repetitively with different tempos or octaves from another string.

Example

```
10 MARY$="GFE-FGGG"
20 PLAY "MB T100 03 L8;XMARY$;P8 FFF4"
30 PLAY "GB-B-4;XMARY$;GFFGFE-."
```

4.99 POINT FUNCTION

Syntax

V=POINT(x,y)

V
Specifies color. Valid returns are 0 through 7 (see Color statement).

x,y
Specify coordinates of a point. Coordinates must be in absolute form (see Chapter 1 for explanation of x and y coordinates).

Purpose

Allows user to read the color of a point from the screen.

Remarks

If you specify a point which is out of range -32768 to 32767, a -1 is returned.

Example

```
10 SCREEN 1
20 FOR C=0 TO 7
30 PSET (10,10),C
40 IF POINT(10,10) <>C THEN PRINT
"Broken Basic!"
50 NEXT C
```

```
10 SCREEN 1
20 IF POINT(i,i) <>0 THEN PRESET (i,i)
ELSE
PSET (i,i)
```

In the second example, BASIC checks the color of a point. If the point is not black, the color changes to black. If the point is black, the color changes to the foreground color. Another way to do this follows:

```
10 SCREEN 1
20 PSET (i,i),1-POINT(i,i)
```

4.100 POKE STATEMENT

Syntax

POKE I,J

where I and J are integer expressions.

Purpose

To write a byte into a memory location.

Remarks

I and J are integer expressions. The expression I represents the address of the memory location and J is the data byte. I must be in the range -32768 to 65535. I is the offset from the current segment, which was set by the last DEF SEG statement (see Section 4.31). For interpretation of negative values of I, see "VARPTR," Section 4.142.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. (See Section 4.97.)

WARNING: Use POKE carefully. If it is used incorrectly, it can cause GW-BASIC to crash.

Example

```
10 POKE &H5A00,&HFF
```

4.101 POS FUNCTION

Syntax	POS(I)
Purpose	To return the current horizontal (column) position of the cursor
Remarks	The leftmost position is 1. I is a dummy argument. To return the current line position of the cursor, use the CSRLIN function. (Section 4.24).
Example	IF POS(X)>60 THEN BEEP

Also see "LPOS Function," Section 4.77.

4.102 PRESET STATEMENT

Syntax

PRESET (x coordinate,y coordinate) [,color]

x coordinate,y coordinate
Sets the point coordinates in either absolute or offset form (see Chapter 1 for explanation of x and y coordinates).

color
Optional. Specifies the color of the point (0-7, see Color statement). If not specified, color is the background color.

Purpose

Sets a point on the screen from which to begin drawing.

Remarks

PRESET and PSET have identical syntaxes. The only difference is that if you do not specify color in PRESET, the background color 0 is selected. In PSET, if you do not specify a color, the color is the foreground color. Line 60 in the example under PSET could be:

```
60 PRESET (i,i)
```

BASIC allows coordinate values to be beyond the edge of the screen and no action is taken nor is an error given. However, values outside the integer range -32768 to 32767 cause an overflow error.

4.103 PRINT STATEMENT

Syntax PRINT [<list of expressions>]

Purpose To output data on the screen.

Remarks If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. GW-BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, GW-BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8 is output as 1E-08. Double precision

numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .000000000-0000001 and 1D-16 is output as 1D-16.

With the interpreter, a question mark may be used in place of the word PRINT in a PRINT statement.

Example 1

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X∧5
30 END
```

will yield

```
 10       0       -25       3125
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2

```
10 INPUT X
20 PRINT X "SQUARED IS" X∧2 "AND";
30 PRINT X "CUBED IS" X∧3
40 PRINT
50 GOTO 10
```

will yield

```
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line. Line 40 causes a blank line to be printed before the next prompt.

Example 3

```
10 FOR X=1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
```

will yield

```
 5  10  10  20  15  30  20  40  25  50
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space.) In line 40, a question mark is used instead of the word PRINT.

4.104 PRINT USING STATEMENT

Syntax

PRINT USING <string exp>;<list of expressions>

Purpose

To print strings or numbers using a specified format.

Remarks/
Examples

<list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons.

<string exp> is a string literal (or variable) composed of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!"

Specifies that only the first character in the given string is to be printed.

"\ n spaces \

"Specifies that 2 + n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
```

50 PRINT USING "\ \";A$;B$;"!!"
will yield
LO
LOOKOUT
LOOK OUT !!

"&" Specifies a variable length string field. When the field is specified with "&", the string is output without modification.

Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
will yield
LOUT
```

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 each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

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, the digit will always be printed (as 0, if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
0.78
```

```
PRINT USING "###.##";987.654
987.65
```

```
PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20    5.30    66.79    0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

\+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

—	A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign. PRINT USING "+##.## ";-68.95,2.4,55.6,-.9 -68.95 +2.40 +55.60 -0.90 PRINT USING "##.##- ";-68.95,22.449,-7.01 68.95- 22.45 7.01-
**	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 more digits. 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 $$. Negative numbers cannot be used unless the minus sign trails to the right. PRINT USING "$$###.##";456.78 $456.78
**$	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 "$. When negative numbers are printed, the minus sign will appear immediately to the left of the dollar sign. PRINT USING "$##.##";2.34 ***$2.34

,

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 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
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-justifies, 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

PRING 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.

% 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 will result.

4.105 PRINT# AND RETURN PRINT# USING STATEMENTS

Syntax

PRINT#<file number>,[USING <string exp>;] <list of expressions>

Purpose

To write data to a sequential file.

Remarks/
Examples

<file number> is the number used when the file was opened for output. <string exp> consists of formatting characters as described in "PRINT USING Statement," Section 4.104. The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

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 the data, so that it will be input correctly.

In the list of expressions, numeric expressions should be delimited by semicolons. For example:

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly 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 read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, 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"

And the statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and "   93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

Note

See also "WRITE# Statement," Section 4.148.

PRINT # [COM 1] in Verbindung mit
FSTCH völlig transparent
Ausnahme: [illegible] (13) (10) und (13)

4.106 PSET STATEMENT

Syntax

PSET (x coordinate,y coordinate) [,color]

x coordinate, y coordinate
Specifies point coordinates in either absolute or offset form (see Chapter 1 for explanation of x and y coordinates).

color
Optional. Specifies the color of the point (0-7, see Color statement). If you do not specify, color is foreground color.

Purpose

Sets a point on the screen from which to begin drawing.

Remarks

Note that BASIC allows coordinate values to be beyond the edge of the screen and no action is taken nor is an error given. However, values outside the integer range -32768 to 32767 cause an overflow error.

Example

If black is set as the background color, lines 10 through 40 of the example draw a diagonal line from point (0,0) to point (100,100). Lines 50 through 70 clear the line by setting each point to 0, black.

```
10 SCREEN 1
20 FOR i=0 TO 100
30 NEXT
40 FOR i=100 TO 0 STEP -1
50 PSET (i,i),0
60 NEXT
```

4.107 PUT STATEMENT

Syntax 1 PUT [#]<file number >[,<record number>]

Purpose To write a record from a random buffer to a random access file.

Remarks <file number> is the number under which the file was opened. If <record number> is omitted, the record will assume the next available record number (after the last PUT). The largest possible record number is 32,767. The smallest record number is 1.

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

See "GET AND PUT STATEMENT" in this chapter, for discussion of PUT with screen capabilities.

Note PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement.

In the case of WRITE#, GW-BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

Syntax 2 PUT (xl,yl),<array name>[,<action verb>

This form of the PUT statement is used with graphics capabilities.

4.108 RANDOMIZE STATEMENT

Syntax RANDOMIZE [<expression>]

Purpose To reseed the random number generator.

Remarks If <expression> is omitted, GW-BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the argument with each run.

Example

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
```

will yield

```
Random Number Seed (-32768 to 32767)? 3
    (user types 3)
```

will yield

```
.885982  .4485668  .586328  .1194246  .7039225
```

```
Random Number Seed (-32768 to 32767)? 4
    (user types 4 for new sequence)
```

will yield

```
.803506  .1625462  .929364  .2924443  .322921
```

```
Random Number Seed (-32768 to 32767)? 3
    (same sequence as first run)
```

will yield

```
.885982  .4845668  .586328  .1194246  .7039225
```

Note that the numbers your program produces may not be the same as the ones shown here.

4.109 READ STATEMENT

Syntax

READ <list of variables>

Purpose

To read values from a DATA statement and assign them to variables. (See "DATA Statement," Section 4.26.)

Remarks

A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an "Out of data" error message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see "RESTORE Statement," Section 4.113)

Example 1

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

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

Example

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

will yield

```
CITY       STATE      ZIP
DENVER,    COLORADO   80211
```

This program reads string and numeric data from the DATA statement in line 30.

4.110 REM STATEMENT

Syntax REM <remark>

Purpose To allow explanatory remarks to be inserted in a program.

Remarks REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into from a GOTO or GOSUB statement. Execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

Important Do not use this in a data statement, because it would be considered legal data.

Example

```
.
.
.
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
.
.
.
```

or

```
.
.
.
120 FOR I=1 TO 20 'CALCULATE AVERAGE
VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
.
.
.
```

4.111 RENUM COMMAND

Syntax RENUM [[<new number>][,[<old number>] [,<increment>]]]

Purpose To renumber program lines.

Remarks <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.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and REL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number in xxxxx" is printed. The incorrect line number reference is not changed by RENUM, but line number yyyyy may be changed.

Note RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples RENUM

Renumbers the entire program. The first new line number will be 10. Lines will be numbered in increments of 10.

RENUM 300,,50

Renumbers the entire program. The first new line number will be 300. Lines will be numbered in increments of 50.

RENUM 1000,900,20

Renumbers the lines from 900 up so they start with line number 1000 and are numbered in increments of 20.

GW-BASIC Compiler

The RENUM command is not supported by the GW-BASIC Compiler.

4.112 RESET COMMAND

Syntax	RESET
Purpose	To close all files on all drives.
Remarks	RESET closes all open files on all drives and writes the directory track to every disk with open files. All files must be closed before a disk is removed from its drive.

4.113 RESTORE STATEMENT

Syntax RESTORE [<line number>]

Purpose To allow DATA statements to be reread from a specified line.

Remarks After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
 .
 .
 .
```

4.114 RESUME STATEMENT

Syntaxes

RESUME

RESUME 0

RESUME NEXT

RESUME <line number>

Purpose

To continue program execution after an error recovery procedure has been performed.

Remarks

Any one of the four syntaxes shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0
  Execution resumes at the statement that caused the error.

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 printed.

Example

```
10 ON ERROR GOTO 900
 .
 .
 .
900 IF (ERR=230)AND(REL=90) THEN
PRINT "TRY
AGAIN":RESUME 80
 .
 .
 .
```

GW-BASIC Compiler

In GW-BASIC Compiler, if an error occurs in a single-line function, RESUME and RESUME NEXT will attempt to resume execution at the line containing the function.

4.115 RETURN STATEMENT

Syntax
RETURN [<line>]

line
Specifies program line number you want to return to.

Purpose
Returns program from a subroutine.

Remarks
The optional line argument has been added to the Return statement to allow non-local returns from event trapping routines. You may want to go back to the BASIC program at a fixed line number while still eliminating the GOSUB entry the trap created. Use RETURN <line> with care! Any other GOSUB, WHILE, or FOR which was active at the time of the trap remains active.

4.116 RIGHT$ FUNCTION

Syntax RIGHT$(X$,I)

Purpose to return the rightmost I characters of string X$.

Remarks If I is equal to the number of characters in X$ (LEN(X$)), returns X$. If I = 0, the null string (length zero) is returned.

Example

```
10 A$="DISK BASIC"
20 PRINT RIGHT$(A$,5)
```

will yield

```
BASIC
```

Also see the LEFT$ and MID$ functions, Sections 4.65 and 4.81, respectively.

4.117 RND FUNCTION

Syntax RND[(X)]

Purpose To return a random number between 0 and 1.

Remarks The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded (see "RANDOMIZE Statement," Section 4.108. However, X 0 always restarts the same sequence for any given X.

X > 0 or X omitted generates the next random number in the sequence. X = 0 repeats the last number generated.

Example

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
will yield
 24 30 31 51 5
```

Note The values produced by the RND function may vary with different implementations of GW-BASIC.

4.118 RUN STATEMENT/COMMAND

Syntax 1 RUN [<line number>]

Purpose To execute the program currently in memory.

Remarks If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. GW-BASIC always returns to command level after a RUN statement is executed.

Example RUN

Syntax 2 RUN <filespec>[,R]

Purpose To load a file from disk into memory and run it.

Remarks The <filespec> must include the filename used when the file was saved. (Your operating system may append a default filename extension if one was not supplied in the SAVE command.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain open.

Example RUN "NEWFIL",R

GW-BASIC Compiler NCR GW-BASIC Compiler supports the RUN and RUN <line number> forms of the RUN statement. GW-BASIC Compiler does not support the "R" option with RUN. If you want this feature, the CHAIN statement should be used.

Note that RUN executes EXE files created by the GW-BASIC Compiler; it does not support the execution of GW-BASIC source files, as does the interpreter.

Other EXE files not created with the GW-BASIC Compiler are also executable with the RUN <filespec> statement. These may be EXE files created in other languages besides GW-BASIC.

4.119 SAVE COMMAND

Syntax

SAVE <filespec>[{,A | ,P}]

Purpose

To save a program file on disk.

Remarks

<filespec> is a quoted string that conforms to your operating system's requirements for filenames. Your operating system will append a default filename extension if one was not supplied in the SAVE command. If a filename already exists, the file will be written over.

The A option saves the file in ASCII format. If the A option is not specified, GW-BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

The P option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples

SAVE "COM2",A

Saves the program COM2 in ASCII format.

SAVE "PROG",P

Saves the program PROG as a protected file which cannot be altered.

GW-BASIC Compiler

The GW-BASIC Compiler does not support the SAVE command.

4.120 SCREEN FUNCTION

Syntax	SCREEN mode.
	Mode must be either a 0 or 1:
	0=text mode 1=graphics mode
Purpose	Selects the software to handle either text or graphics mode. (See Chapter 1 for an explanation of screen modes.)
Remarks	If the value of the mode parameter is valid, the software stores the new screen mode, and erases the screen. It does not change the foreground and background colors.
	If the parameter value is invalid (not 0, 1, or 2 — 2 for compatibility and conversion), an "Illegal Function Call" message is displayed. The screen mode remains as it was before the statement was entered.
	If the new screen mode is the same as the previous mode, the software only erases the screen.
Examples	10 SCREEN 0 Selects text mode 20 SCREEN 1 Switches to graphics mode

4.121 SCREEN STATEMENT

Syntax SCREEN [<spec1>][,<spec2>]...

Purpose To set the specifications for the display screen.

4.122 SGN FUNCTION

Syntax SGN(X)

Purpose To indicate the value of X, relative to zero:

If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example

```
ON SGN(X)+2 GOTO 100,200,300
```

Branches to 100 if X is negative, 200 if X is 0, and 300 if X if positive.

4.123 SIN FUNCTION

Syntax SIN(X)

Purpose To return the sine of X, where X is in radians.

Remarks COS(X) = SIN(X+3.14159/2).

Example PRINT SIN(1.5)
will yield
.9974951

See also "COS Function," Section 4.22.

4.124 SOUND STATEMENT

Syntax

SOUND <frequency,duration>

frequency
Specifies desired frequency in Hertz (cycles per second). Enter the desired number from 220 to 32767.(See also table of notes and frequencies.)

duration
Specifies desired length of the sound measured in clock ticks. (1 clock tick = 55 ms.) Enter the number of clock ticks. (See also table of typical tempos.)

Purpose

Generates sound through the speaker.

The following table correlates notes with their frequencies. The tuning note A has a frequency of 440.

Note	Freq.	No.*	Note	Freq.	No.*
Pause	32767		F#	740	43
A	220	22	G	784	44
A#	233	23	G#	830	45
B	247	24	A	880	46
C	262	25	A#	930	47
C#	277.2	26	B	987.8	48
D	293.6	27	C	1046.4	49
D#	311.6	28	C#	1106	50
E	329.6	29	D	1174.6	51
F	349.2	30	D#	1244	52
F#	370	31	E	1318.6	53
G	392	32	F	1397	54
G#	416	33	F#	1480	55
A	440	34	G	1568	56
A#	466	35	G#	1660	57
B	493.2	36	A	1760	58
**C	523.2	37	A#	1864	59
C#	554.8	38	B	1975.6	60
D	587.4	39	C	2093	61
D#	622	40	C#	2217.4	62
E	659.2	41	D	2349.4	63
F	698.4	42			

* See the Play statement for use of these numbers.
** Middle C

Remarks

The Sound statement produces a sound that continues until another Sound statement is reached. If a Sound statement with a duration of 0 is encountered, any currently running Sound statement is turned off. (If no Sound statement is running, SOUND freq,0 has no effect.)

You can cause sounds to be buffered so program execution does not stop when a new Sound statement is encountered. (See the MB command explained under the Play statement.)

To create periods of silence, use SOUND 32767, duration.

The duration for one beat is calculated from beats per minute. Divide the beats per minute into 1092 (the number of clock ticks in a minute). The following table shows typical tempos in terms of clock ticks (duration).

	Tempo	Beats/ Minute	Ticks/ Beat (Duration)
very slow	Larghissimo		
	Largo	40-60	27.3-18.2
	Larghetto	60-66	18.2-16.55
	Grave		
	Lento		
↓	Adagio	66-76	16.55-14.37
slow	Adagietto		
↓	Andante	76-108	14.37-10.11
medium	Andantino	108-120	10.11-9.1
↓	Moderato		
fast	Allegretto		
	Allegro	120-168	9.1-6.5
	Vivace		
	Veloce		
↓	Presto	168-208	6.5-5.25
very fast	Prestissimo		

Example

The following program creates a glissando up and down.

```
10 FOR I=220 TO 2200 STEP 20
20 SOUND I, 0.5
30 NEXT
40 FOR I=2200 TO 220 STEP -20
50 SOUND I, 0.5
60 NEXT
```

4.125 SPACE$ FUNCTION

Syntax SPACE$(X)

Purpose To return a string of spaces of length X.

Remarks The expression X is rounded to an integer and must be in the range 0 to 255.

Example

```
10 FOR I=1 TO 5
20 X$=SPECE$(I)
30 PRINT X$,I
40 NEXT I
will yield
 1
  2
   3
    4
     5
```

Also see "SPC Function," Section 4.126.

4.126 SPC FUNCTION

Syntax SPC(I)

Purpose To skip spaces in a PRINT statement. I is the number of spaces to be skipped.

Remarks SPC may only be used with PRINT and LPRINT statements. I must be in range 0 to 255. A';' is assumed to follow the SPC(I) command.

Example PRINT "OVER" SPC(15) "THERE"
will yield
OVER THERE

Also see "SPACE$ Function," Section 4.125.

4.127 SQR FUNCTION

Syntax SQR(X)

Purpose To return the square root of X.

Remarks X must be > = 0.

Example

```
10 FOR X=10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
```

will yield

```
 10         3.162278
 15         3.872984
 20         4.472136
 25         5
```

4.128 STICK FUNCTION

Syntax

x = STICK(n)

x is a numeric variable for storing the result of the function.

(n) is a numeric expression returning an unsigned integer in the range 0 to 3.

Purpose

To return the x and y coordinates of the two joysticks.

Remarks

The values returned for n can be:

0 — returns the x coordinate for joystick A. Also stores the x and y values for both joysticks for the following function calls:

1 — Returns the y coordinate of joystick A.

2 — Returns the x coordinate of joystick B.

3 — Returns the y coordinate of joystick B.

Example

```
10 CLS
20 LOCATE 1,1
30 PRINT "X=%";STICK(0)
40 PRINT "Y=5";STICK(1)
50 GOTO 20
```

This example creates an endless loop to display the value of the x,y coordinate for joystick A.

4.129 STOP STATEMENT

Syntax STOP

Purpose To terminate program execution and return to command level.

Remarks 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

With GW-BASIC Interpreter, the STOP statement does not close files. With GW-BASIC Compiler, all open files are closed.

GW-BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command. (see Section 4.21).

Example

```
10 INPUT A,B,C
20 K=A∧2*5.3L=B∧3/.26
30 STOP
40 M=C*K+100:PRINT M
will yield
? 1,2,3
BREAK IN 30

PRINT L
 30.76923
CONT
 115.9
```

Note that because the CONT command is included here, this particular example works only with the interpreter.

GW-BASIC Compiler

If the /D, /E, or /X compiler switches are turned on, the STOP message prints the line number at which execution has stopped.

4.130 STR$ FUNCTION

Syntax STR$(X)

Purpose To return a string representation of the value of X.

Example

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB
30,100,200,300,400,500
.
.
.
```

Also see "VAL Function, Section 4.141.

4.131 STRIG STATEMENT/FUNCTION

Syntax

STRIG ON
STRIG OFF
STRIG STOP
x = STRIG(n)

where x is a numeric variable for storing the result of the function.

(n) is a numeric expression returning an unsigned integer in the range 0 to 3, designating which trigger is to be checked.

Purpose

The STRIG ON statement enables event trapping of joystick activity.

The STRIG OFF statement disables event trapping of joystick activity.

The STRIG STOP statement disables event trapping of joystick activity, but if the joystick is pressed, that event will be remembered and will be trapped as soon as event trapping is enabled.

The x=STRIG(n) function returns the status of a specified joystick trigger.

Remarks

The STRIG ON statement enables joystick event trapping by an ON STRIG statement (see "ON STRIG Statement," Section 4.91). While trapping is enabled, and if a non-zero line number is specified in the ON STRIG statement, GW-BASIC checks between every statement to see if the joystick trigger has been pressed.

The STRIG OFF statement disables event trapping. If an event occurs (i.e., if the trigger is pressed), it will not be remembered.

The STRIG STOP statement disables event trapping, but if an event occurs it will be remembered, and the event trap will take place as soon as trapping is enabled.

In the X = STRIG(n) function, the values returned for (n) can be:

0 — Returns -1 if trigger A was pressed since the last STRIG(0) statement; returns 0 if not.

1 — Returns -1 if trigger A is currently down, 0 if not.

2 — Returns -1 if trigger B was pressed since the last STRIG(2) statement, 0 if not.

3 — Returns -1 if trigger B is currently down, 0 if not.

When a joystick event trap occurs, that occurrence of the event is destroyed. Therefore, the x=STRIG(n) function will always return false inside a subroutine, unless the event has been repeated since the trap. So if you wish to perform different procedures for various joysticks, you must set up a different subroutine for each joystick, rather than including all the procedures in a single subroutine.

Example

```
10 IF STRIG(0) THEN BEEP
20 GOTO 20
```

In this example an endless loop is created to beep whenever the trigger button on joystick 0 is pressed.

GW-BASIC Compiler

See compiler note under "ON STRIG Statement," Section 4.91.

4.132 STRING$ FUNCTION

Syntaxes	STRING$(I,J) STRING$(I,X$)
Purpose	To return a string of length I whose characters all have ASCII code J or the first character of X$.
Example	10 X$=STRING$(10,45) 20 PRINT X$ "MONTHLY REPORT" X$ will yield ----------MONTHLY REPORT----------

4.133 SWAP STATEMENT

Syntax

SWAP <variable>,<variable>

Purpose

To exchange the values of two variables.

Remarks

Any type variable may be swapped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

If the second variable is not already defined when SWAP is executed, an "Illegal function call" error will result.

Example

```
10 A$=" ONE " : B$=" ALL"
   : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
will yield
 ONE FOR ALL
 ALL FOR ONE
```

4.134 SYSTEM COMMAND

Syntax SYSTEM

Purpose To close all open files and return control to the operating system.

Remarks When a SYSTEM command is executed, a "warm start" is performed (i.e., all open files are closed, and the operating system is reloaded without deleting any existing programs or memory except GW-BASIC itself).

GW-BASIC Compiler The GW-BASIC Compiler does not support this command.

4.135 TAB FUNCTION

Syntax TAB(I)

Purpose To move the print postion to I.

Remarks If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example

```
10 PRINT "NAME" TAB(25) "AMOUNT" :
PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES", "$25.00"
```

will yield

```
NAME          AMOUNT

G. T. JONES   $25.00
```

4.136 TAN FUNCTION

Syntax TAN(X)

Purpose To return the tangent of X. X should be given in radians.

Remarks With the interpreter, if TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example

```
10 Y=Q*TAN(X)/2
```

4.137 TIME$ STATEMENT

Syntax

TIME$=<string expression>

<string expression> returns a string in one of the following forms:

hh (sets the hour; minutes and seconds default to 00)

hh:mm (sets the hour and minutes; seconds default to 00)

hh:mm:ss (sets the hour, minutes, and seconds)

Purpose

To set the time. This statement complements the TIME$ function, which retrieves the time.

Remarks

A 24-hour clock is used; 8:00 p.m., therefore, would be entered as 20:00:00.

Example

```
10 TIME$="08:00:00"
```

The current time is set at 8:00 a.m.

4.138 TIME$ FUNCTION

Syntax	TIME$
Purpose	To retrieve the current time. (To set the time, use the TIME$ statement, described in Section 4.137.)
Remarks	The TIME$ function returns an eight-character string in the form hh:mm:ss, where hh is the hour (00 through 23), mm is minutes (00 through 59), and ss is seconds (00 through 59). A 24-hour clock is used; 8:00 p.m., therefore, would be shown as 20:00:00.
Example	10 PRINT TIME$ Prints the time, calculated from the time set with the TIME$ statement.

4.139 TRON/TROFF STATEMENTS/COMMANDS

Syntax

TRON

TROFF

Purpose

To trace the execution of program statements.

Remarks

As an aid in debugging, the TRON statement (executed 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

```
TRON

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
```

will yield

```
[10][20][30][40] 1  10  20
[50][60][30][40] 2  20  30
[50][60][70]
```

```
TROFF
```

Note that this example is for the interpreter only.

GW-BASIC Compiler

In order to use TRON/TROFF, the compiler debug switch /D must be turned on. Otherwise, TRON and TROFF are ignored and a warning message is generated.

4.140 USR FUNCTION

Syntax

USR[<digit>][(<argument>)]

where <digit> specifies which USR routine is being called. See the DEF USR statement, Section 4.32, for rules governing <digit>. If <digit> is omitted, USR0 is assumed.

<argument> is the value passed to the subroutine. It may be any numeric or string expression.

Purpose

To call an assembly language subroutine.

Remarks

In this implementation, if a segment other than the default segment (data segment DS) is to be used, a DEF SEG statement must be executed prior to a USR function call. The address given in the DEF SEG statement determines the segment address of the subroutine.

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

Example

```
100 DEF SEG=&H8000
110 DEF USR0=0
120 X=5
130 Y = USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed.

GW-BASIC Compiler

The USR function is implemented in the compiler to call machine language subroutines. However, there is no way to pass parameters with the USR function, except by using POKE statements to protected memory locations that are later accessed by the machine language routine.

If this method is used, the USR function must preserve the values of all registers except BX. The USR function must return the integer result in the BX register.

There are two alternatives to using POKE statements to pass parameters:

1. If the machine language routine is short enough, it can be stored by making a string containing the ASCII values corresponding to the hexadecimal values of the routine. Use the CHR$ function to insert ASCII values in the string. The start of the routine can then be found by using the VARPTR function. For example, for the string A$, VARPTR(A$) will return the address of the low byte of the string length. The next address contains the high byte of the string length. The next two addresses are: first, the least significant byte, and second, the most significant byte, of the actual address of the string.

   This setup of the string space differs from that of the interpreter. Thus, to find the actual starting address of the routine, use the following instructions:

   ```
   10 A$ = "String containing routine"
   20 I% = VARPTR(A$)
   30 AD = PEEK(I% +3) * 256 +
   PEEK(I% + 2)
   40 REM AD is the start address
   50 of the routine
   ```

   String contents move around in the string space, so any absolute references must be adjusted to reflect the current memory location of the routine.

2. A better alternative is to use MS-Macro Assembler to assemble your subroutines. The subroutines can be linked directly to the compiled program and referenced using the CALL statement.

4.141 VAL FUNCTION

Syntax

VAL(X$)

Purpose

To return the numerical value of string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example,

VAL(" -3")

returns -3.

Example

```
10 READ NAME$,CITY$,STATE$,ZIP$
IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699
    THEN PRINT NAME$ TAB(25) "OUT OF
    STATE"
30 IF VAL(ZIP$)>=90801 AND
    VAL(ZIP$)<=90815
    THEN PRINT NAME$ TAB(25) "LONG
    BEACH"
 .
 .
 .
```

See the STR$ function, Section 4.130, for details on numeric-to-string conversion.

4.142 VARPTR FUNCTION

Syntax 1 VARPTR(<variable name>)

Syntax 2 VARPTR(#<file number>)

Purpose Syntax 1

Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array). For string variables, the address of the first byte of the string descriptor is returned (see "Assembly Language Subroutines," in the NCR GW-BASIC User's Guide for discussion of the string descriptor). The address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

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.

Note All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Syntax 2

For sequential files, returns the starting address of the disk I/O buffer assigned to <file number>. For random files, returns the address of the FIELD buffer assigned to <file number>.

Example

```
100 X=USR(VARPTR(Y))
```

4.143 VARPTR$ FUNCTION

Syntax VARPTR$(<variable name>)

where <variable name> is the name of a variable in the program.

Purpose To return a character form of the memory address of the variable.

Remarks VARPTR$ is primarily used with the DRAW and PLAY statements (Sections 4.35 and 4.98 respectively) in programs that will be compiled.

A value must be assigned to <variable name> prior to execution of VARPTR$. Otherwise, an "Illegal function call" error results. Any type variable (numeric, string, or array) may be used.

VARPTR$ returns a three-byte string in the form:

byte 0 = type
byte 1 = low byte of address
byte 2 = high byte of address

Note, however, that the individual parts of the string are not considered characters.

Note Because array addresses change whenever a new variable is assigned, always assign all simple variables before calling VARPTR$ for an array element.

Example

```
10 PLAY "X"+VARPTR$(A$)
```

Uses the subcommand X, plus the address of A$, as the string expression in the PLAY statement.

4.144 WAIT STATEMENT

Syntax WAIT <port number>,I[,J]

where I and J are integer expressions.

Purpose To suspend program execution while monitoring the status of a machine input port.

Remarks 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 exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, GW-BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

Warning It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine. To avoid this, WAIT must have the specified value at <port number> during some point in the program execution.

Example 100 WAIT 32,2

4.145 WHILE...WEND STATEMENTS

Syntax

WHILE <expression>
.
.
[<loop statements>]
.
.
WEND

Purpose

To execute a series of statements in a loop as long as a given condition is true.

Remarks

If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. GW-BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If is is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement a "WEND without WHILE" error.

Example

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU
    LOOP
110 WHILE FLIPS
115     FLIPS=0
120     FOR I=1 TO J-1
130         IF A$(I)>A$(I+1) THEN
                SWAP A$(I),A$(I+1):FLIPS=1
140     NEXT I
150 WEND
```

Note

Do not direct program flow into a WHILE/WEND loop without entering through the WHILE statement.

GW-BASIC Compiler

With GW-BASIC Compiler, WHILE/WEND constructions must be statically nested, even within FOR/NEXt or other WHILE/WEND loops. Static nesting means that each WHILE/WEND pair cannot reside partly in, and partly outside, the nesting pair. For example, the following construction is not allowed:

```
FOR I = 1 to 10
    A = COUNT
    WHILE A = 1

NEXT I
    A = A — 1
    WEND
```

4.146 WIDTH STATEMENT

Syntax

WIDTH <file number>,<size>
WIDTH <device>,<size>
WIDTH <size>

file number
Numeric expression in the integer range 1 to 255. This is the number of a file opened.

size
Numeric expression in the integer range 1 to 255. This is the new width.

device
String expression which identifies the device. Valid devices are SCRN: and LPT1:.

Purpose

Sets the printed line width in number of characters.

Remarks

WIDTH <file number>,<size>
If the file is open to LPT1:, the line printer's printed line width is immediately changed to the new size specified. This statement allows you to change the width at will while the file is open. This form of the WIDTH statement has meaning only for LPT1:.

WIDTH "LPT1:",<size>
Used as a deferred width assignment for the line printer, this form of the WIDTH statement stores the new width value without actually changing the current width setting. A subsequent OPEN"LPT1:" FOR OUTPUT AS <number> will use the new size specified while the file is open.

WIDTH <size>
or
WIDTH "SCRN:",<size>
This command has no effect because the NCR DECISION MATE V always has a screen width of 80 columns. However, NCR GW-BASIC will

accept this command without displaying an error message.

If you enter any value outside the range 1 to 255 for width or file number, an "Illegal Function Call" error occurs. The width or file number remains as it was before the illegal value was entered.

No data is lost by using the WIDTH statement. The software simply adds a carriage return after sending the indicated number of characters. For example, if you have a 60-character line and a 40-character printer, and if you issue WIDTH 40, the first 40 characters will be printed on one line and the next 20 characters on the next line.

4.147 WRITE STATEMENT

Syntax

WRITE [<list of expressions>]

Purpose

To output data to the screen.

Remarks

If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output to the screen. The expressions in the list may be numeric and/or string expressions. They must be separated by commas.

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, GW-BASIC inserts a carriage return/linefeed.

WRITE outputs numeric values using the same format as the PRINT statement. (See Section 4.103.)

Example

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
will yield
 80, 90,"THAT's ALL'
```

4.148 WRITE# STATEMENT

Syntax

WRITE#<file number>,<list of expressions>

Purpose

To write data to a sequential file.

Remarks

<file number> is the number under which the file was OPENed in "O" mode (see "OPEN Statement," Section 4.92). The expressions in the list are string or numeric expressions. They must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to the file and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/linefeed sequence is inserted after the last item in the list is written to the file.

Example

Let A$="CAMERA" and B$="93604-1"

The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT$ statement, such as

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

Error Codes And Error Messages

GW-BASIC error messages include:

Runtime error messages
Compiler invocation error messages
Compiletime error messages
MS-LINK error messages

The compiler invocation and compiletime error messages apply only to GW-BASIC Compiler.

A.1 RUNTIME ERROR MESSAGES

Code	Number	Message
NF	1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
SN	2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.). With GW-BASIC, the incorrect line will be part of a DATA statement. GW-BASIC Interpreter automatically enters edit mode at the line that caused the error.

RG	3	Return without GOSUB
		A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
OD	4	Out of data
		A READ statement is executed when there are no DATA statements with unread data remaining in the program.
FC	5	Illegal function call
		A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of:

1. A negative or unreasonably large subscript.
2. A negative or zero argument with LOG.
3. A negative argument to SQR.
4. A negative mantissa with a non-integer exponent.
5. A call to a USR function for which the starting address has not yet been given.
6. An improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.
7. A negative record number used with GET or PUT.
8. Concatenation of strings that creates a string greater than 32767 characters in length (compiler error).

OV	6	Overflow The result of a calculation is too large to be represented in GW-BASIC number format. If underflow occurs, the result is zero and execution continues without an error.
OM	7	Out of memory A program is too large, or has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated for a file buffer to be allocated.
UL	8	Undefined line A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN... ELSE, or DELETE statement.
BS	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.
DD	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.
/0	11	Division by zero A division by zero is encountered in an expression; or, the operation of involution results, in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

		With the compiler, this error may occur if the integer -32768 is divided by 1 or -1, or if -32768 is MODed by 1 or -1.
ID	12	Illegal direct
		A statement that is illegal in direct mode is entered as a direct mode command.
TM	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.
OS	14	Out of string space
		String variables have caused BASIC to exceed the amount of free memory remaining. GW-BASIC will allocate string space dynamically, until it runs out of memory.
LS	15	String too long
		An attempt is made to create a string more than 255 characters long.
ST	16	String formula too complex
		A string expression is too long or too complex. The expression should be broken into smaller expressions.
CN	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.

UF	18	Undefined user function
		A USR function is called before the function definition (DEF statement) is given.
	19	No RESUME
		An error handling routine is entered but contains no RESUME statement.
	20	RESUME without error
		A RESUME statement is encountered before an error handling routine is entered.
	21	Unprintable error
		An error message is not available for the 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
		Occurs if one of the signals to be tested (CTS, DSR, or CD) is missing when a file is opened or if the host computer loses CTS, DSR, or CD while waiting to put data in the output buffer.
	25	Device fault
		Occurs if host computer loses DSR or CD.

26 FOR without NEXT

A FOR statement was encountered without a matching NEXT.

27 Out of paper

The printer device is out of paper.

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.

Disk Errors

50 Field overflow

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error

An internal malfunction has occurred in GW-BASIC.

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, or OPEN statement/command references a file that does not exist on the current disk.

54 Bad file mode

An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN statement with a file mode other than I, O, or R.

With the compiler, this error may also occur when an attempt is made to read from a file opened for output or appending.

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 for overrun, parity, or framing errors in communication. It is a fatal error; i.e., the operating system 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.

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 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 and 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 255 directory entries are full.

68 Device Unavailable

An attempt was made to open a file to a non-existent device. It may be that hardware did not exist to support the device, such as LPT2: or LPT3:, or was disabled. This occurs if an OPEN COM1 statement is executed but RS-232 support was disabled via the /C:0 switch directive on the command line.

69 Communication Buffer Overflow

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

Increase the size of the communications receive buffer via the /C:switch.

Implement an XON/XOFF protocol with the host/satellite to turn transmit off long enough for characters in the input buffer to be processed.

Use a lower baud rate for transmit and receive.

70 Disk Write Protect

Occurs when an attempt is made to write to a disk that is write-protected. Use an ON ERROR GOTO statement to recover.

71 Disk not ready

Could be caused by a number of problems. The most likely is that the disk is not inserted properly.

72 Disk Media Error

Occurs when the FDC controller detects a hardware or media fault. This usually indicates damaged media. Copy any existing files to a new disk and reformat the damaged disk. FORMAT flags the bad tracks and places then in a bad-track file. The remainder of the disk is now usable.

74 Rename across disks

An attempt was made to rename a file with a new drive designation. This is not allowed.

The following error messages are received only with GW-BASIC Compiler. They are "severe" errors which cannot be trapped.

Cannot find A:BASRUNG.EXE
Enter new drive letter:

This message appears if the runtime module is not available in the default drive or in drive A:. The second line prompts for input of the correct drive letter.

Internal Error — No Line Number

Occurs when the error address cannot be found in the line number table during error trapping. This occurs if there are no integer line numbers between 0 and 65527. It may also occur if the line number table has been accidentally overwritten by the user program.

Internal Error — String Space Corrupt

This occurs when an invalid string in string space is being deallocated, usually in a string assignment statement. See the listing following the next error message, "Internal Error — String Space Corrupt during G.C." for additional causes.

Internal Error — String Space Corrupt during G.C.

This occurs when an invalid string in string space is being deleted during garbage collection. (G.C. stands for garbage collection.)

The probable causes for either of the "String Space Corrupt" errors are:

1. A string descriptor or string back pointer has been improperly modified. This may occur if you use an assembly language subroutine to modify strings.

2. Out-of-range array subscripts are used and string space is inadvertently modified. The /D switch may be used to ensure that array subscripts do not exceed the array bounds.

3. Improper use of the POKE and/or DEF SEG statements that may modify string space improperly.

4. Mismatched COMMOM declarations between two chained programs.

5. Line Input [illegible] String-Länge > 254.

Error in EXE file

Occurs when a file is not of the correct type. It must be an executable file if it is to be executed with RUN or CHAIN.

Program too large

Not enough memory is available to load BASRUNG.EXE.

A.2 COMPILER INVOCATION ERROR MESSAGES

Invocation errors occur when illegal input is given on the command line or in response to prompts during invocation. The messages that may occur when the compiler is invoked are listed below:

Bad filename

File specification has been entered incorrectly.

Bad switch: /<s>

Illegal compiler switch <s>

Can't create file

Disk is write protected or the disk is full.

Command error: '<c>'

An error has occurred at the character specified by the character <c>

Disk <d> full

The disk in the drive <d> has no more directory entries. If no <d> appears, the disk in the default drive is full.

File not found

The file does not exist on the specified disk.

A.3 COMPILETIME ERROR MESSAGES

For errors that occur at compiletime, the compiler outputs the line containing the error, an arrow beneath that line pointing to the place in the line where the error occurred, and a two-character code for the error. In some cases, the compiler reads ahead on a line to determine whether an error has actually occurred. In those cases, the arrow points a few characters beyond the error, or to the end of the line. The GW-BASIC compiletime errors described below are divided into severe errors and warning errors. When a severe error occurs, the compiler will attempt to continue so that any additional errors will also be detected. However, in general, the resulting code will not be usable. On the other hand, when a warning error occurs, compilation continues, but warning errors are printed on the screen to point out poorly constructed program statements.

A.3.1 Severe Errors

Severe compiletime errors are indicated either by a long message or by a two-letter code. Long messages are described first.

Long Messages

Long errors describe general conditions that are not associated with a particular line number.

Binary source file

The file you have attempted to compile is not an ASCII file. All source files SAVEd from within the Interpreter should be saved with the ",A" option.

Internal error

An internal error has occurred in the Compiler.

Line <n> is undefined

A GOTO or GOSUB statement refers to a nonexistent line number.

Memory overflow

Available memory has been exhausted. Try compiling with the /S switch or without any of the debug switches. If memory is still exhausted, break your program into parts and use the CHAIN command.

Missing NEXT for variable

No NEXT was found for a FOR statement.

Two-letter Codes

Code	**Meaning**
BS	Bad Subscript Illegal dimension value Wrong number of subscripts
CD	Duplicate COMMON variable
CN	COMMON array not dimensioned
CO	COMMON out of order
DD	Array Already Dimensioned More than one DIM statement for same array DIM statement after initial use of array OPTION BASE after array dimensioned
FD	Function Already Defined
FN	FOR/NEXT Error FOR loop index variable already in use FOR without NEXT NEXT without FOR
IN	INCLUDE Error $INCLUDE file not found

LL	Line Too Long
LS	String Constant Too Long
OM	Out of Memory

Array too big
Data memory overflow
Too many statement numbers
Program memory overflow

OV	Math Overflow
SN	Syntax error — caused by one of the following:

Illegal argument name
Illegal assignment target
Illegal constant format
Illegal debug request
Illegal DEFxxx character specification
Illegal expression syntax
Illegal function name
Illegal function formal parameter
Illegal separator
Illegal format for statement number
Invalid character
Missing AS
Missing equal sign
Missing GOTO or GOSUB
Missing comma
Missing INPUT
Missing line number
Missing left parenthesis
Missing minus sign
Missing operand in expression
Missing right parenthesis
Missing semicolon
Name too long
Expected GOTO or GOSUB
String assignment required
String expression required
String variable required
Illegal syntax

Variable required
Wrong number of arguments
Formal parameters must be unique
Single variable only allowed
Missing TO
Illegal FOR loop index variable
Illegal COMMON name
Missing THEN
Missing BASE
Illegal subroutine name

SQ Sequence Error

Duplicate statement number
Statement out of sequence

TC Too Complex

Expression too complex
Too many arguments in function call (limit of 60)
Too many dimensions (limit of 255)
Too many variables for LINE INPUT (limit of 1)
Too many variables for INPUT (limit of 60)

TM Type Mismatch

Data type conflict
Variable must be of same type

UC Unrecognizable Command
Statement unrecognizable
Command not implemented

UF Function Not Defined

WE WHILE/WEND Error

WHILE without WEND
WEND without WHILE

/0 Division by Zero

Also occurs if the integer -32768 is divided by 1 or -1, or if -32768 is MODed by 1 or -1.

/E	Missing "/E" Switch
/X	Missing "/X" Switch

A.3.2 Warning Errors

Warning errors will not terminate a compilation. However, they often indicate a situation in which a program will not operate as intended. The warning messages are:

CODE	*MESSAGE*
MC	Metacommand error
ND	Array not Dimensioned
SI	Statement Ignored

Statement ignored
Unimplemented command

A.4 MS LINK ERROR MESSAGES

A listing of MS-LINK error messages may be found in the manuals that are supplied with your MS-DOS software. For your convenience, we have also listed them here.

All errors cause the link session to abort. After the cause has been found and corrected, MS-LINK must be rerun. The following error messages are displayed by MS-LINK.

Attempt to access data outside of segment bounds, possibly bad object module

There is probably a bad object file.

Bad numeric parameter

Numeric value is not in digits.

Cannot open temporary file

MS-LINK is unable to create the file VM.TMP because the disk directory is full. Insert a new disk. Do not remove the disk that will receive the LIST.MAP file.

Error: DUP record too complex

DUP record in assembly language module is too complex. Simplify DUP record in assembly language program.

Error: Fixup offset exceeds field width

An assembly language instruction refers to an address with a short instruction instead of a long instruction. Edit assembly language source and reassemble.

Input file read error

There is probably a bad object file.

Invalid object module

An object module(s) is incorrectly formed or incomplete (as when assembly is stopped in the middle).

Symbol defined more than once

MS-LINK found two or more modules that define a single symbol name.

Program size or number of segments exceeds capacity of linker

The total size may not exceed 384K bytes and the number of segments may not exceed 255.

Requested stack size exceeds 64K

Specify a size greater than or equal to 64K bytes with the STACK switch.

Segment size exceeds 64K

64K bytes is the addressing system limit.

Symbol table capacity exceeded

Very many and/or very long names were entered, exceeding the limit of approximately 25K bytes.

Too many external symbols in one module

The limit is 256 external symbols per module.

Too many groups

the limit is 10 groups.

Too many libraries specified

The limit is 8 libraries.

Too many PUBLIC symbols

The limit is 1024 PUBLIC symbols

Too many segments or classes

The limit is 256 (segments and classes taken together).

Unresolved externals: <list>

The external symbols listed have no defining module among the modules of library files specified

VM read error

This is a disk error; it is not caused by MS-LINK.

Warning: No stack segment

None of the object modules specified contains a statement allocating stack space, but the user typed the STACK switch.

Warning: Segment of absolute or unknown type

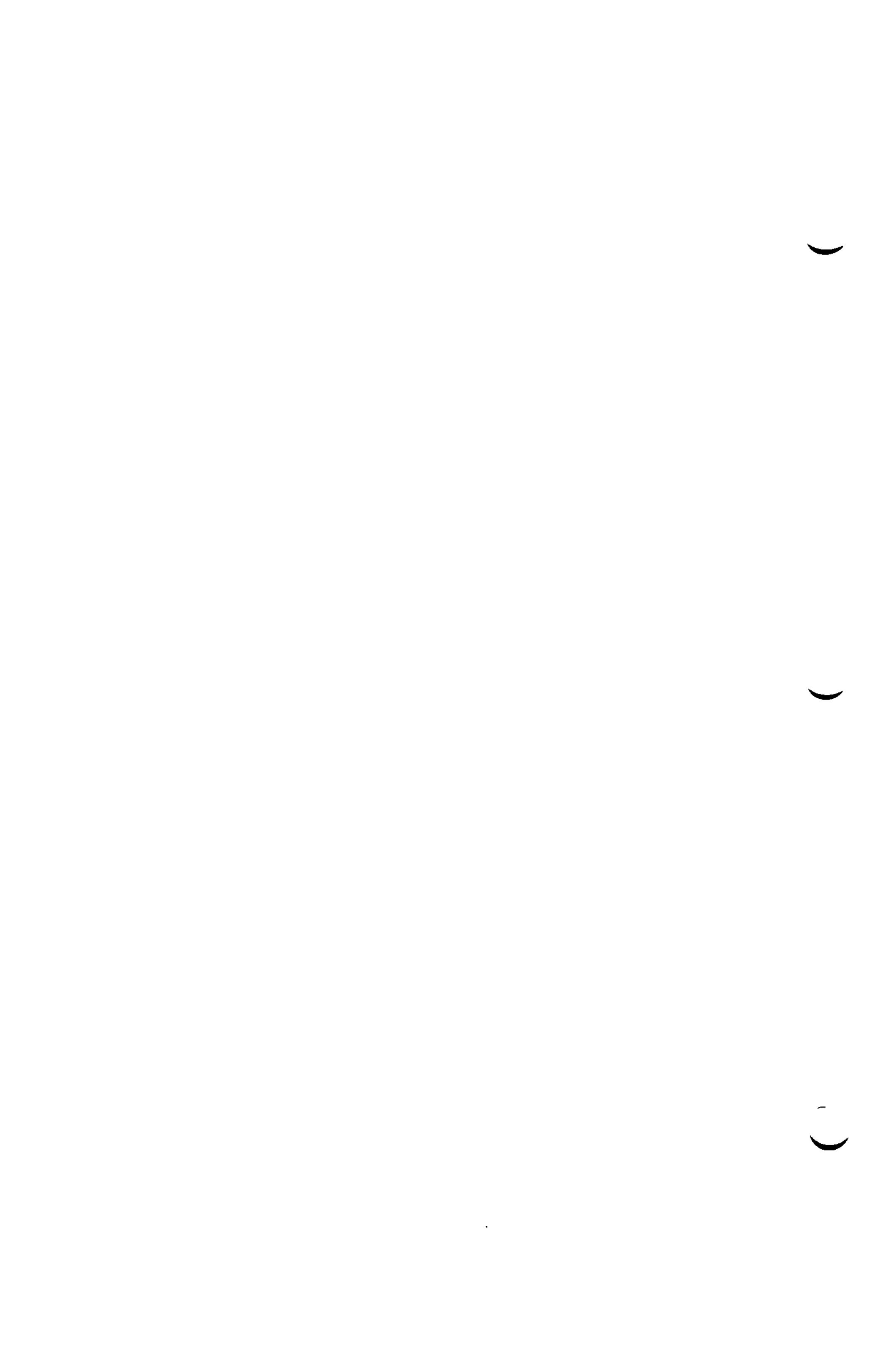
There is a bad object module or an attempt has been made to link modules that MS-LINK cannot handle (e.g., an absolute object module).

Write error in TMP file

No more disk space remains to expand VN.TMP file.

Write error on run file

Usually, there is not enough disk space for the run file.

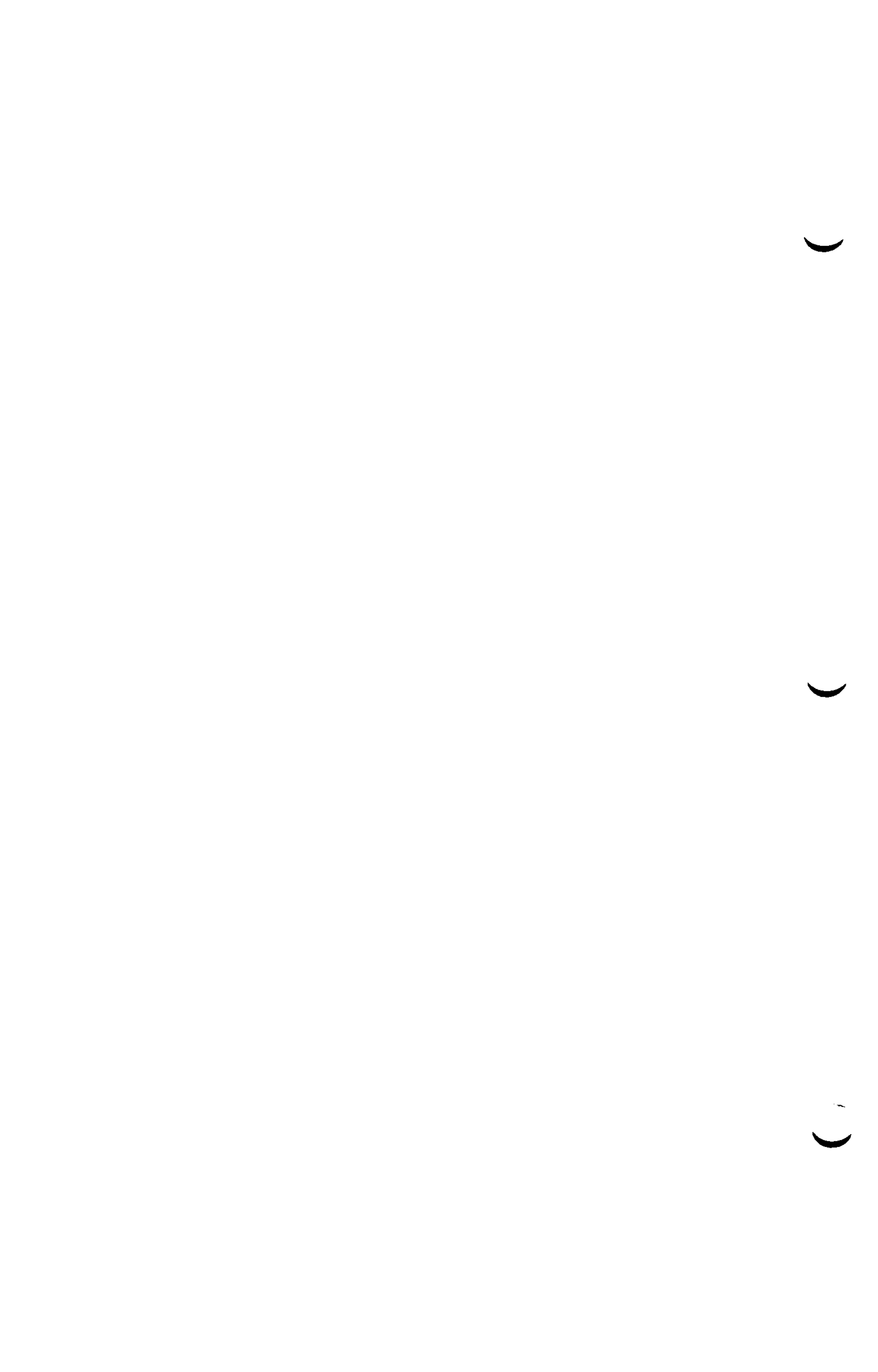


Mathematical Functions

Derived Functions

Functions that are not intrinsic to Microsoft GW-BASIC may be calculated as follows.

Function	**Microsoft GW-BASIC Equivalent**
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(−X*X+1))
INVERSE COSINE	ARCCOS(X)=−ATN(X/SQR(−X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X−1)) +SGN(SGN(X)−1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X−1)) +(SGN(X)−1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)−EXP(−X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(−X))/2
HYPERBOLIC TANGENT	TANH(X)=(EXP(X)−EXP(−X))/ (EXP(X)+EXP(−X))
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(−X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)−EXP(−X))
HYPERBOLIC COTANGENT	COTH(X)=(EXP(X)+EXP(−X))/ (EXP(X)−EXP(−X))
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X−1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1−X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(−X*X+1)+1j)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1) +1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X−1))/2



ASCII Character Codes

Dec	Hex	CHR
000	00H	NUL
001	01H	SOH
002	02H	STX
003	03H	ETX
004	04H	EOT
005	05H	ENQ
006	06H	ACK
007	07H	BEL
008	08H	BS
009	09H	HT
010	0AH	LF
011	0BH	VT
012	0CH	FF
013	0DH	CR
014	0EH	SO
015	0FH	SI
016	10H	DLE
017	11H	DC1
018	12H	DC2
019	13H	DC3
020	14H	DC4
021	15H	NAK
022	16H	SYN
023	17H	ETB
024	18H	CAN
025	19H	EM
026	1AH	SUB
027	1BH	ESCAPE
028	1CH	FS
029	1DH	GS
030	1EH	RS
031	1FH	US
032	20H	SPACE
033	21H	!
034	22H	"
035	23H	#
036	24H	$
037	26H	&
038	27H	'
043	2BH	+
044	2CH	,
045	2DH	−
046	2EH	.
047	2FH	/
048	30H	0
049	31H	1
050	32H	2
051	33H	3
052	34H	4
053	35H	5
054	36H	6
055	37H	7
056	38H	8
057	39H	9
058	3AH	:
059	3BH	;
060	3CH	<
061	3DH	=
062	3EH	>
063	3FH	?
064	40H	@
065	41H	A
066	42H	B
067	43H	C
068	44H	D
069	45H	E
070	46H	F
071	47H	G
072	48H	H
073	49H	I
074	4AH	J
075	4BH	K
076	4CH	L
077	4DH	M
078	4EH	N
079	4FH	O
081	51H	Q
082	52H	R
086	56H	V
087	57H	W
088	58H	X
089	59H	Y
090	5AH	Z
091	5BH	[
092	5CH	\
093	5DH	[
094	5EH	∧
095	5FH	
096	60H	¯,
097	61H	a
098	62H	b
099	63H	c
100	64H	d
101	65H	e
102	66H	f
103	67H	g
104	68H	h
105	69H	i
106	6AH	j
107	6BH	k
108	6CH	l
109	6DH	m
110	6EH	n
111	6FH	o
112	70H	p
113	71H	q
114	72H	r
115	73H	s
116	74H	t
117	75H	u
118	76H	v
119	77H	w
120	78H	x
121	79H	y
122	7AH	z
124	7CH	
125	7DH	

039	28H	(	083	53H	S	126	7EH	~
040	29H	)	084	54H	T	127	7FH	DEL
041	2AH	*	085	55H	U			
042								

Dec = decimal, Hex = hexadecimal (H), CHR = character,
LF = Linefeed, FF = Formfeed, CR = Carriage Return,
DEL = Rubout

GW-BASIC Reserved Words

The following is a list of reserved words used in GW-BASIC.

ABS
AND
ASC
ATN
AUTO
BEEP
BLOAD
BSAVE
CALL
CDBL
CHAIN
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COLOR
COM
COMMON
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DATE$
DEFDBL
DEFINT
DEFSNG
DEFSNG
DEFSTR
DEF FN
DEF USR
DELETE
DIM
DRAW
EDIT
ELSE
END
EOF
ERASE
ERL
ERR
ERROR
END
EXP
FIELD
FILES
FIX
FOR
FRE
GET
GPSUB
HEX$
IF
IMP
INP
INPUT
INKEY$
INPUT#
INPUT$
INSTR
INT
KEY
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET
MERGE
MID$
MKD
MKI$
MKS$
MOD
MOTOR
NAME
NEW
NEXT
NOT
OCT$
ON
OPEN
OPEN COM
OPTION
OR
PAINT
PALETTE
PALETTE USING
PEEK
PEN
PLAY
POINT
POKE
POS
PRESET
PRINT
PRINT# USING
PSET
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RIGHT$
RND
RSET
RUN
SAVE
SBN
SIN
SOUND
SPACE
SPC
SQR
STICK
STOP
STR$
STRIG
STRING$
SWAP
SYSTEM
TAB
TAN
THEN
TIME$
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VARPTR$
WAIT
WEND
WHILE
WIDTH
WRITE
WRITE#
XOR

INDEX

ABS function, 4-4
Active page, 3-2
Addition, 3-12
Arctangent, 4-6
Arithmetic operators, 3-11
Array variables, 3-8, 4-33, 4-51
Arrays, 3-8, 4-26, 4-33, 4-60
ASC function, 4-5
ASCII
    codes, C-1
    format, 4-5, 4-18, 4-124
Assembly language subroutines, 4-13, 4-49, 4-210, 4-213
ATN function, 4-6
AUTO command, 4-7

BACKSPACE editor function, 2-5
BEEP statement, 4-8
BLOAD statement, 4-9
Boolean operators, 3-16
BREAK editor function, 2-5
BSAVE statement, 4-11

CALL statement, 4-13
CALLS statement, 4-15
CARRIAGE RETURN editor function, 2-4
CDBL function, 4-16
CHAIN statement, 4-17, 4-30
Character set, 3-2
CHR$ function, 4-21
CINT function, 4-22
CIRCLE statement, 4-23
CLEAR LOGICAL LINE key, 2-5
CLEAR SCREEN editor function, 2-5

CLEAR statement, 4-26
CLOSE statement, 4-28
CLS statement, 4-29
Color selection, 1-4
COLOR statement, 4-30
COM as event specifier, 1-7
COM statement, 4-32
COM trapping, 1-7
Command
    definition, 4-1
Command level, 3-1
COMMON statement, 4-33
Communications, 1-7
Compiler invocation error messages, A-11
Compiler severe errors, A-13
Compiletime error messages, A-13
Concatenation, 3-18
Constants
    defined, 3-4, 3-5
    numeric, 3-5
    string, 3-5
CONT command, 4-36, 4-111
Continuation of a line, 3-2
Control characters, 3-4
COS function, 4-37
CSNG function, 4-38
CSRLIN function, 4-39
CURSOR DOWN editor function, 2-4
CURSOR HOME editor function, 2-4
CURSOR LEFT editor function, 2-4
CURSOR position, 2-4 to 2-6
CURSOR RIGHT editor function, 2-4
CURSOR UP editor function, 2-4
CVD function, 4-40
CVI function, 4-40
CVS function, 4-40

DATA statement, 4-41, 4-172
DATE$ function, 4-43
DATE$ statement, 4-42
Declaration characters, 3-7
DEF FN statement, 4-44
DEF SEG statement, 4-48

DEF USR statement, 4-49, 4-210
Default device, 3-2
DEFDBL statement, 3-8, 4-46
DEFINT statement, 3-8, 4-46
DEFSNG statement, 3-8, 4-46
DEFSTR statement, 3-8, 4-46
DELETE command, 4-50
DELETE editor function, 2-1
Device-independent I/O, 1-8
DIM statement, 4-51
Direct mode, 2-2, 3-1, 4-84, 4-133,
Display page, 3-2
Division, 3-12
Double precision, 3-6, 4-16, 4-46, 4-157
DRAW statement, 4-53

EDIT command, 2-1, 3-1, 4-56
Editing programs, 2-1
Editor, 2-1
Editor functions, 2-4 to 2-6
END statement, 4-57
EOF function, 4-58
ERASE statement, 4-60
ERL variable, 4-61
ERR variable, 4-61
Error codes, 3-19, 4-61, 4-62, A-1
Error handling, 4-61, 4-62, 4-133
Error messages, 3-19, A-1 to A-19
ERROR statement/command, 4-62
Error trapping, 4-61, 4-179
Errors - Warning, A-17
Escape, 3-3
Evaluation of operators
    arithmetic, 3-11
    logical, 3-16
Event trapping, 1-7
EXP function, 4-64
Exponentiation, 3-13
Expressions, 3-11

FIELD statement, 4-65
Files

protected, 4-186
random, 4-65, 4-74, 4-100, 4-116, 4-123, 4-127, 4-139, 4-170
sequential, 4-58, 4-91, 4-100, 4-112, 4-116, 4-139, 4-166, 4-221
FILES statement, 4-68
Filespec, definition of, 4-2
FIX function, 4-69
FOR...NEXT statement, 4-70
FRE function, 4-73
Full screen editor, 2-1
advantages, 2-2
cursor position, 2-4
Function, definition of, 4-1
Function key display, 2-5
Functional operators, 3-18
Function of special keys, 2-4
Functions, 4-44

GET statement, 4-74
GET and PUT statements, 4-75
GOSUB statement, 4-80
GOTO statement, 4-82
Graphics, 1-1
Graphics mode, 1-2

HEX$ function, 4-83
Hexadecimal, 3-6, 4-83

IF...GOTO statement, 4-84
IF...THEN statement, 4-84
IF...THEN...ELSE statement, 4-84
Indirect mode, 3-1, 2-2
INKEY$ function, 4-87
INP function, 4-88
INPUT statement, 4-36, 4-65, 4-89
INPUT# statement, 4-91
INPUT$ function, 4-92
INSERT editor function, 2-5
INSTR function, 4-93
INT function, 4-94
Integer, 4-22, 4-69, 4-94
Integer division 3-12 to 3-14

Joystick, 1-7

KEY as event specifier, 1-7
KEY statement, 4-95
KEY trapping, 1-7
KEY (n) statement, 4-198
KILL statement, 4-100

LEFT$ function, 4-105
LEN function, 4-106
LET statement, 4-107
Line continuation, 3-2
Line editing, 2-1
Line format, 3-1
LINE INPUT statement, 4-111
LINE INPUT# statement, 4-112
Line length, 3-2
Line number generation, 4-7
Line numbers, 3-1, 3-2
Line printer, 4-112 to 4-114, 4-218
LINE statement, 4-108
LIST statement, 3-2, 4-113
LCOPY statement, 4-101
LOAD command, 4-115, 4-186
LOC function, 4-116
LOCATE statement, 4-117
LOF function, 4-119
LOG function, 4-120
Logical line, 2-2
Logical line definition with INPUT, 2-1
Logical operators, 3-16 to 3-18
Loops, 4-70, 4-216
LPOS function, 4-121
LPRINT statement, 4-122
LPRINT USING statement, 4-122
LSET statement, 4-123

Mathematical functions, B-1
MERGE command, 4-124
MID$ function, 4-126
MID$ statement, 4-125
MKD$ function, 4-127

MKI$ function, 4-127
MKS$ function, 4-127
MOD operator, 3-13
Modes of operation, 3-1
Modulus arithmetic, 3-13
MS-LINK error messages, A-17
Multiplication, 3-12
Music, 1-6

NAME statement, 4-128
Negation, 3-12
NEW command, 4-129
NEXT WORD editor function, 2-4
Numeric constants, 3-5
Numeric variables, 3-7

OCT$ function, 4-130
Octal, 3-6, 4-130
ON COM statement, 4-131
ON ERROR GOTO statement, 4-133
ON GOSUB, in event trapping, 1-7
ON GOSUB statement, 4-134
ON GOTO statement, 4-134
ON KEY statement, 4-135
ON STRIG statement, 4-137
OPEN BASE statement, 4-145
OPEN COM statement, 4-141
OPEN statement, 4-139
Operators, 3-12, 3-16 to 3-18
    Boolean, 3-16
    functional, 3-18
    string, 3-18
OPTION BASE statement, 4-145
Order of evaluation
    arithmetic operators, 3-11
    logical operators, 3-16
OUT statement, 4-146
Overflow, 3-14, 4-64, 4-206

PAINT statement, 4-147
PEEK function, 4-149, 4-154
Peripherals support, 1-7

Pixels, 1-2
PLAY statement, 4-150
POINT function, 4-153
POKE statement, 4-154
POS function, 4-155
Precedence
    arithmetic operators, 3-11
    logical operators, 3-16
PRESET statement, 4-156
PREVIOUS word editor function, 2-4
PRINT statement, 4-157
PRINT USING statement, 4-160
PRINT# statement, 4-166
PRINT# USING statement, 4-166
Protected files, 4-186
PSET statement, 4-169
PUT statement, 4-65, 4-170

Random files, 4-65, 4-74, 4-100, 4-116, 4-123, 4-127, 4-139, 4-170
Random numbers, 4-171, 4-183
RANDOMIZE statement, 4-171, 4-183
READ statement, 4-172
Relational operators, 3-15
REM statement, 4-174
RENUM command, 4-17, 4-61, 4-175
Reserved words, D-1
RESET command, 4-177
RESTORE statement, 4-178
RESUME statement, 4-179
RETURN,
    in event trapping, 1-7
RETURN statement, 4-80, 4-181
RETURN PRINT$, 4-166
RIGHT$ function, 4-182
RND function, 4-183
RSET statement, 4-123
RUN command, 4-184
RUN statement/command, 4-184
Runtime error messages, A-1

SAVE command, 4-115, 4-186
SCREEN function, 4-187

Screen modes, 1-1
SCREEN statement, 4-188
Sequential files, 4-58, 4-91, 4-100, 4-112, 4-116, 4-139, 4-166, 4-221
Severe errors, compiler, A-12
SGN function, 4-189
SIN function, 4-190
Single precision, 3-6, 4-38, 4-46, 4-157
Sound, 1-6
SOUND statement, 4-191
Space requirements for variables, 3-9
SPACE$ function, 4-194
SPC function, 4-195
Special characters, 3-2
Special keys, 2-4
SQR function, 4-196
Statement, definition of, 4-1
STICK function, 4-197
STOP statement, 4-36, 4-57, 4-80, 4-198
STR$ function, 4-199
STRIG
    as event specifier, 1-7
    function, 4-200
    statement, 4-200
    trapping, 1-7
String and numeric constants, 3-5
String functions, 4-40, 4-93, 4-105, 4-126, 4-182, 4-199, 4-212
String operators, 3-18
String space, 4-26, 4-73
String variables, 3-7, 4-46, 4-111
STRING$ function, 4-202
Subroutines, 4-13, 4-80, 4-134
Subscripts, 3-8, 4-51, 4-145
Subtraction, 3-12
SWAP statement, 4-203
SYSTEM command, 4-204

Tab, 3-4
TAB editor function, 2-6
TAB function, 4-205
TAN function 4-206
Text mode, 1-1
TIME$ function, 4-208
TIME$ statement, 4-207

Transcendental functions, 3-12
TROFF statement/command, 4-209
TRON statement/command, 4-209
Type conversion, 3-9

USR function, 4-49, 4-210

VAL function, 4-212
Variables, 3-7
    array, 4-30, 4-51
    order of, 4-33
    passing with COMMON, 4-18
    string, 4-46, 4-111 to 4-112
Variables in edited lines, 2-2
VARPTR function, 4-213
VARPTR$ function, 4-214
Visual page, 3-2

WAIT statement, 4-215
Warning errors, A-17
WEND statement, 4-216
WHILE statement, 4-216
WIDTH statement, 4-218
WRITE statement, 4-220
WRITE# statement, 4-221
Writing programs, 2-1

X and Y coordinates, 1-3

GW™-BASIC Compiler User's Guide

For MS™-DOS

COPYRIGHT NOTICE

Copyright© 1983 by Microsoft Corporation, all rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a 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 the prior written permission of Microsoft Corporation.

TRADEMARKS

Microsoft and the Microsoft logo are registered trademarks of Microsoft Corporation. MS, GW, Music Macro Language, and Graphics Macro Language are trademarks of Microsoft Corporation. Teletype is a registered trademark of Teletype Corporation.

DISCLAIMER OF WARRANTY

NCR Corporation and Microsoft Corporation make no representations or warranties with respect to the contents hereof and specifically disclaim any implied warranties of merchantability or fitness for any particular purpose. Further, NCR Corporation and Microsoft Corporation reserve the right to revise this publication and to make changes from time to time in the content hereof without obligation to notify any person or organization of such revisions or changes.

The GW-BASIC Compiler Software and Manual are sold AS IS and without warranty as to performance. While NCR Corporation and Microsoft Corporation firmly believe this to be a high quality product, the user must assume all risks of using the program.

INTRODUCTION

NCR GW-BASIC Compiler is an optimizing compiler designed to complement NCR GW-BASIC Interpreter. GW-BASIC Compiler allows you to create programs that in most cases:

1. Execute faster than the same interpreted programs.

2. Require less memory than the same interpreted programs.

3. Provide source-code security.

These benefits can be critical in:

Graphics applications. Execution speed can often make or break an application.

Business applications. Several chained programs can be supported by a main menu.

Commercial applications. Software is sold in a competitive marketplace and source-code security is essential.

Another major benefit is that GW-BASIC Compiler has been created to support most of the interpreted GW-BASIC language. Thus, the interpreter and the compiler complement each other, providing you with an extremely powerful GW-BASIC programming environment. In this environment, you can quickly run and debug a program from within the GW-BASIC Interpreter, and then later compile the same program to increase its speed of execution and decrease its space in memory.

An additional GW-BASIC Compiler feature is a runtime module named BASRUNG.EXE. This module contains most of the facilities you need to run a program. The runtime module is loaded when program execution begins, and you can later run a chained program without reloading. This allows you to save disk space by developing a system of related programs that can all be run using the same runtime environment. For example, with a system of four programs, you can save at least 48K of disk space.

Note that language, operational, and implementation differences between GW-BASIC Compiler and GW-BASIC Interpreter are described in Chapter 9, "A Compiler/Interpreter Comparison". Review the information in that chapter before compiling any of your

programs, even those that already run without problems under the GW-BASIC Interpreter.

System Requirements

The minimum memory requirements for GW-BASIC Compiler running under the MS™-DOS operating system is 128K.

One disk drive is required. We recommend two disk drives, however, for easier operation.

Use of GW-BASIC Compiler requires the Microsoft LINK Linking Loader, (MS™-LINK). MS-LINK is a standard MS-DOS utility.

HOW TO USE THIS MANUAL

The NCR GW-BASIC Compiler User's Guide is designed for users who are not familiar with the compiler as a programming tool. Therefore, this User's Guide provides a step-by-step introduction to the GW-BASIC Compiler and its use. However, it also contains the technical information needed for more advanced compiling, linking, and execution of programs.

This manual assumes that the user has a working knowledge of the GW-BASIC language. For reference information, consult the NCR GW-BASIC References Manual.

This User's Guide contains the following information:

Introduction

Describes the NCR GW-BASIC Compiler and provides a Syntax Notation for your reference.

Chapter 1 Introduction to Compilation

Introduces you to the vocabulary associated with compilers and presents an overview of program development with GW-BASIC Compiler.

Chapter 2 Demonstration Run

Takes you step by step through compiling, linking, and running a demonstration program.

Chapter 3 Editing a Source Program

Describes how to create a GW-BASIC source program for later compilation.

Chapter 4 Debugging With GW-BASIC Interpreter

Describes how to debug your BASIC source file with GW-BASIC Interpreter.

Chapter 5 Compiling

Gives you technical details about using GW-BASIC Compiler, including descriptions of the command line syntax and the compiler options.

Chapter 6 Linking

Describes how to use Microsoft LINK Linking Loader (MS-LINK) to link your programs to runtime support. (See the "Important" note following this listing.)

Chapter 7 Running a Program

Describes how to run your final executable program.

Chapter 8 Metacommands

Describes the metacommands that are used to control source listing file information. Also describes use of the $INCLUDE metacommand, which lets you switch source files during compilation.

Chapter 9 A Compiler/Interpreter Comparison

Describes the operational and implementation differences between the NCR GW-BASIC Compiler and the GW-BASIC Interpreter. It is important to study these differences and to make the necessary editing changes in your GW-BASIC program before you use the compiler.

Chapter 10 Communications

Describes the GW-BASIC statements required to support RS-232 asynchronous communication with other computers and peripherals.

The appendices show you how to create a system of programs with the runtime module, provide memory and segment maps, describe the use of assembly language subroutines, and explain how to use disk files.

Important

> *The NCR GW-BASIC Compiler User's Guide* occasionally refers to the MS-DOS utilities, MS-LINK and MS™-MACRO. In addition to the information provided in this manual, these utilities are described in the manuals that are supplied with your MS-DOS software.

Syntax Notation

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user-entered data. When the angle brackets enclose lowercase text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose uppercase text, the user must press the key named by the text; for example, <RETURN>.
{ }	Braces indicate that the user has 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.
¦	Vertical bars separate choices within braces. At least one of the entries separated by bars must be chosen unless the entries are also enclosed in square brackets.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered, exactly as shown.

GW-BASIC Compiler User's Guide

Contents

Introduction
System Requirements ii
How to Use This Manual ii
Syntax Notation iv

Chapter 1 Introduction to Compilation
1.1 Compilation vs. Interpretation 1-1
1.2 Vocabulary 1-2
1.3 The Program Development Process 1-4

Chapter 2 Demonstration Run

Chapter 3 Editing a Source Program
3.1 $Include Metacommand 3-1
3.2 Compiler/Interpreter Differences 3-2
3.3 Line Length 3-3

Chapter 4 Debugging With GW-BASIC Interpreter

Chapter 5 Compiling
5.1 Default File Specifications 5-1
5.2 Compiler Invocation 5-2
5.3 Compiler Switches 5-5
5.4 Configuring for GW-BASIC 5-12

Chapter 6 Linking
6.1 Linking to BASRUNG.LIB Runtime Library . . . 6-1
6.2 Linking to BASCOMG.LIB Runtime Library . . . 6-3

Chapter 7 Running a Program

Chapter 8 Metacommands
8.1 Syntax 8-1
8.2 Description 8-2

Chapter 9 A Compiler/Interpreter Comparison
9.1 Operational Differences 9-1
9.2 Implementation Differences 9-1

Chapter 10 Communications
10.1 Opening a Communications File 10-1
10.2 Communication I/O 10-1
10.3 Control Signals 10-3
10.4 Sample Program 10-4

Appendix A Creating a System of Programs With the Runtime Module

Appendix B Source Listing Format

Appendix C Memory Maps

Appendix D Runtime Segment Maps

Appendix E Assembly Language Subroutines

Introduction to Compilation

1.1 COMPILATION vs. INTERPRETATION

A microprocessor can execute only its own machine instructions; it cannot execute GW-BASIC statements directly. Therefore, before a program can be executed, statements in a GW-BASIC program must be translated into the machine language of your NCR Decision Mate V. Compilers and interpreters are two types of programs that perform this translation. This discussion explains the difference between these two translators.

1.1.1 INTERPRETATION

NCR GW-BASIC Interpreter translates your GW-BASIC program line-by-line *at runtime*. To execute a GW-BASIC statement, the interpreter analyzes the statement, checks for errors, then performs the GW-BASIC function requested.

If a statement must be executed repeatedly (inside a FOR/NEXT loop, for example), this translation process must be repeated each time the statement is executed.

A GW-BASIC program is stored as a list of numbered lines; the lines are not available as absolute memory addresses during interpretation. Therefore, branches such as GOTO and GOSUB statements cause the interpreter to examine the line numbers in the list, starting with the first, until the referenced line is found.

Similarly, a list of all variables is maintained by the interpreter. When a reference to a variable is made in a GW-BASIC statement, this list must be searched from the beginning until the referenced variable is found. Thus, absolute memory addresses are not associated with the variables in your program.

1.1.2 COMPILATION

A compiler, on the other hand, translates a source program and creates a new file called an object file. The object file contains relocatable machine code (see Section 1.2, "Vocabulary," for defini-

tions). All translation takes place before runtime; no translation of your GW-BASIC source file occurs during the execution of your program. In addition, absolute memory addresses are associated with variables and with the targets of GOTO and GOSUB statements, so that lists of variables or of line numbers does not have to be searched during execution of your program.

GW-BASIC Compiler is an optimizing compiler. Optimizations such as reordering expressions and eliminating subexpressions increase the speed of execution and decrease the size of your program.

These factors combine to measurably increase the execution speed of your program. The amount of execution time you save with the compiler depends on the makeup of your program. If your program includes a great deal of input/output or many floating-point calculations, for example, it may not run noticeably faster with the compiler. But in most cases, execution of compiled GW-BASIC programs is 3 to 10 times faster than execution of the same program under the interpreter. If maximum use of integer variables is made, execution can be up to 30 times faster.

1.2 VOCABULARY

This section reviews some of the vocabulary that is commonly used when discussing compilers.

First, a GW-BASIC program is commonly called a GW-BASIC "source." The source file is the input file to the compiler. It must be in ASCII format. The compiler translates this source and creates, as output, a new file called a "relocatable object" file. These two files have the default extensions .BAS and .OBJ, respectively.

Other terms that you will see in this manual are related to stages in the development and execution of a compiled program. These stages are listed below.

Compile time — The time during which the compiler is executing, and during which it compiles a GW-BASIC source file and creates a relocatable object file.

Link time — The time during which MS-LINK is executing, and during which it loads and links together relocatable object files and library files.

Runtime — The time during which a compiled and linked program is executing. By convention, runtime refers to the execution time of your

program rather than to the execution time of the compiler or the linker.

You should also be familiar with the following linking and runtime terms.

Module — A fundamental unit of code. There are several types of modules, including relocatable and executable modules. The compiler creates relocatable modules that are later manipulated by MS-LINK. Your final executable program is an executable module.

Executable — A module is executable if the code within it is in a form that can be used, without further translation, by the computer.

Relocatable — A module is relocatable if the code within it can be placed and run at different locations in memory. The relocatable modules created by the compiler are an intermediate stage between source code and executable code; they are changed into executable modules by MS-LINK.

Global reference — A variable name or label in a given module that is referenced by a routine in another module. Global labels are entry points into modules.

Unbound global reference — A global reference in a module that is not declared in that module. MS-LINK tries to resolve this situation by searching for the declaration of that reference in other modules. If such a declaration is found in a module, that module is loaded into memory (if it is not yet in memory) and becomes part of your load file. These other modules are usually library modules in the runtime library.

If the variable or label is found, the address associated with it is substituted for the reference in the first module, and is then said to be "bound." When a variable is not found in any module, it is said to be "undefined."

Routine — Executable code residing in a module. More than one routine may reside in a module. The runtime module contains a majority of the library routines needed to implement the GW-BASIC language. A library routine usually corresponds to a feature or subfeature of the GW-BASIC language.

Runtime support — The body of routines that may be linked to your compiled .OBJ file. These routines implement various features of the GW-BASIC language. The BASCOMG.LIB, BASRUNG.LIB, and the runtime module all contain runtime support routines. See Chapter 6, "Linking", for more information on runtime support.

Runtime module — A module containing most of the routines needed to implement the GW-BASIC language. It is a peculiarity of the runtime module that it is an executable .EXE file. The runtime module is named BASRUNG.EXE. The runtime module is, for the most part, a library of routines: it is made executable so that you can see the version number of the module.

BASRUNG.LIB runtime library — A few modules used to load BASRUNG.EXE at runtime and to move segments around in memory to permit chaining.

BASCOMG.LIB runtime library — A collection of modules containing routines almost identical in function to similar routines contained in the runtime module. However, this library does not support COMMOM between CHAINed subprograms. It does support a version of CHAIN that is equivalent to the simple RUN <filename> command. See Chapter 6, "Linking", for discussion of the runtime libraries.

Linking — The process in which the linker (MS-LINK) computes absolute addresses for labels and variables in relocatable modules, and then resolves all global references by searching the BASRUNG.LIB or BASCOMG.LIB runtime library. After loading and linking, MS-LINK saves the modules that it has loaded into memory as a single .EXE file on your disk. This entire process is called linking.

Complete understanding of all the above terms is not essential for continued reading. You may want to refer back to these terms later, as you become familiar with the compiler and with MS-LINK.

1.3 THE PROGRAM DEVELOPMENT PROCESS

This discussion of the process development process is keyed to Figure 1. Use the figure for reference when reading this text. The development process described here uses the BASRUNG.EXE runtime module and the BASRUNG.LIB runtime library.

Program development begins with (1) the creation of a GW-BASIC source file. The best way to create a GW-BASIC source file is with the editing facilities of GW-BASIC Interpreter, although you can use any general purpose text editor if you wish. Note that files must be SAVEd from GW-BASIC with the ,A option.

We recommend that you use GW-BASIC Interpreter (2) to debug your programs by running them to check for syntax and program logic errors. There are a few differences in the languages understood by the compiler and the interpreter, but for the most part they are identical. Because of this similarity, running a program with the interpreter runs a much quicker syntactic and semantic check of your program than compiling, linking, and finally executing it.

After you have debugged your program with the interpreter, or if you do not have GW-BASIC Interpreter, (3) compile the program to check out differences that may exist between interpreted and compiled GW-BASIC. The compiler flags all syntax errors as it reads your source file. If compilation is successful, the compiler creates a relocatable .OBJ file.

The .OBJ file is not executable, or needs to be linked to the BASRUNG.LIB or BASCOMG.LIB runtime library. You may want to include your own assembly language routines to increase the speed of execution of a particular algorithm, or to handle more complex microprocessor operations. For these cases, use the Microsoft MACRO Assembler (4) to assemble routines that you may later link to your program. See the description of MS-MACRO in the manuals that are supplied with your MS-DOS software for more information.

Microsoft LINK Linking Loader (5) links all modules needed by your program, and produces as output an executable object file with .EXE as the default extension. This file can be (6) executed like any .EXE file by simply entering the file's base name (the file name less its .EXE extension).

This program development process is demonstrated in the following chapter, Chapter 2, "Demonstration Run."

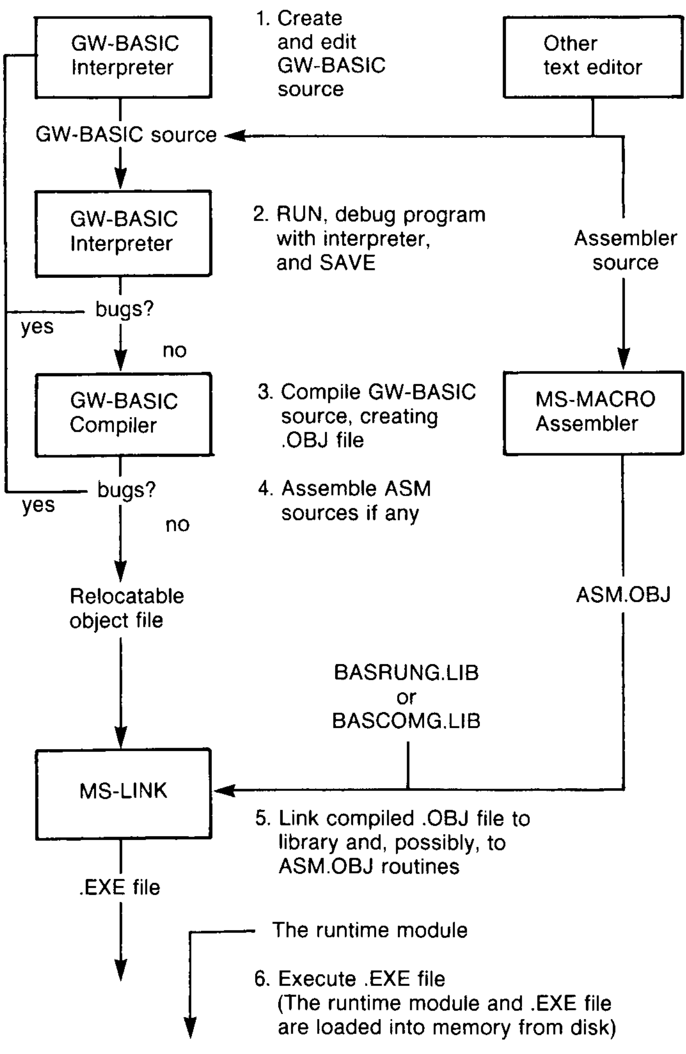


Figure 1 The Program Development Process

Demonstration Run

This chapter provides step-by-step instructions for GW-BASIC Compiler. We strongly recommend that you compile the demonstration program before compiling any other programs. For more technical information, read Chapters 3 through 9 of this manual. If you enter commands exactly as described in this chapter, you should have a successful session with GW-BASIC Compiler. If a problem does arise, carefully perform each step again.

IMPORTANT:
Before you begin this demonstration run, make back-up copies of your disks. After making copies, copy the operating system onto each one, so that each disk can be used to start up the operating system. In addition, copy the MS-DOS utility file, LINK.EXE, along with BASCOMG.LIB and BASRUNG.LIB, to a blank, formatted system disk. These three files will be used when linking your program.

The five steps in developing a program with GW-BASIC Compiler are:

1. Editing (entering and correcting the GW-BASIC program).

2. Debugging with the interpreter (using GW-BASIC Interpreter to run your program; this step is optional).

3. Compiling (creating a relocatable object file).

4. Linking (creating an executable program).

5. Running (executing the program).

Because we have prepared a debugged demonstration program on disk, you do not have to perform the first two steps in the program development process. Therefore, the demonstration run begins with compilation. Note that we have SAVEd the demonstration program on disk with the ,A option, since all files must be in ASCII format to be readable by the compiler.

In the following procedure, two disk drives are assumed. They are called A: and B:, and the procedure begins with A: as the default drive. In general, GW-BASIC Compiler system disks are in drive A:, and a single work disk in drive B: contains all user created files.

In the following discussion, all prompts are in bold face type. User-entered data are in regular face type. The symbol <RETURN> is indicated where only a carriage return is entered. Otherwise, carriage returns are assumed at the end of each user-entered line.

To create an executable compiled program, take the following steps:

1. Start up your system.

   With the disk containing GWBCOM.COM in drive A: and an empty work disk in drive B:, start up your system.

2. Log on to drive B:.

   From A:, log onto drive B: by entering:

   A> B:

   Now B: is the default drive. All files that are created will be placed there unless specified otherwise.

3. Create a GW-BASIC source file.

   GW-BASIC programs can be created with any available text editor. However, for this demonstration run we will use the program provided on disk, DEMO.BAS. For consistency, GW-BASIC source files should always be given the .BAS extension.

4. Invoke the compiler by entering:

   B> A:GWBCOM

5. Input filenames

   The compiler then prints an informative heading and prompts you for the name of your GW-BASIC source program.

   Enter:

   Source filename [.BAS]: A:DEMO

   The default extension .BAS is assumed. Drive A: is specified because DEMO.BAS is located there. After you have entered the name of a legal filename for the source file, you are prompted to enter the name of the relocatable object file that you want to create.

Enter:

Object filename [DEMO.OBJ]: DEMO

The default name is enclosed in brackets in the prompt. This default name can be selected by simply entering a <RETURN>. It can also be selected by entering DEMO as we have done.

The object file is placed on drive B: because we may later have to swap disks in drive A:. Remember that all files created in this demonstration run are written to B:.

The final prompt is for the name of the source listing file. The source listing file is created during compilation, and lists your GW-BASIC source and any compilation errors or warnings as they occur. By default, the listing file is sent to the null file, NUL, and no file is created. Writing to the NUL file is equivalent to not writing a file at all. However, error messages are always displayed on the screen.

To specify this default, you need only enter <RETURN>. However, if any part of a file specification is entered after the prompt, the default extension is .LST and the default device is the currently logged drive. For this demonstration, we want to send this file to the console.

Therefore, we enter:

Source listing [NUL.LST]: CON

After you have completed your input, compilation begins and the source listing file is sent to the console screen as the source file is read.

6. Look for error messages.

   As your program is compiled, error messages are displayed on the terminal screen. For the demonstration program, there should be no error messages displayed. When the compiler has finished, it displays the message:

```
20574  Bytes Available
17726  Bytes Free

    0  Warning Error(s)
    0  Severe Error(s)
```

(The number of bytes available and bytes free varies with a particular system.) Program control is then returned to the operating system.

At this point, you should see one new file (DEMO.OBJ) listed in the B: directory:

7. Link routines in the runtime library to your .OBJ file.

   After compilation, you are ready to link your program. To do this, make sure that a disk containing MS-LINK and the BASRUNG.LIB runtime library is in drive A:. (See the "Important" note at the beginning of this chapter if you do not have such a disk.)

   To begin linking, enter:

   ```
   B: A:LINK
   ```

   MS-LINK prompts you for the name of your relocatable object file:

   ```
   Object Modules [.OBJ]: DEMO
   ```

   The .OBJ extension is assumed for the object file. You are next prompted for the names of the run file and the linker list file. Simply enter <RETURN> after each prompt to specify the default files given in brackets:

   ```
   Run File [B:DEMO.EXE]: <RETURN>
   List File [B:DEMO.MAP]: <RETURN>
   ```

Note that .MAP is the default extension for the linker's list file.

The next prompt is for the library that you wish to link. Several options are available at this point, but for this demonstration run, we will indicate default values for all remaining parameters. This is done by entering a semi-colon.

Therefore, when the "Libraries" prompt appears, enter:

```
Libraries [.LIB]:A:;
```

The result is that MS-LINK searches the BASRUNG.LIB library and invokes the BASRUNG.EXE runtime module.

The run file DEMO.EXE will rely on the runtime module BASRUNG.EXE for all routines that it needs at runtime.

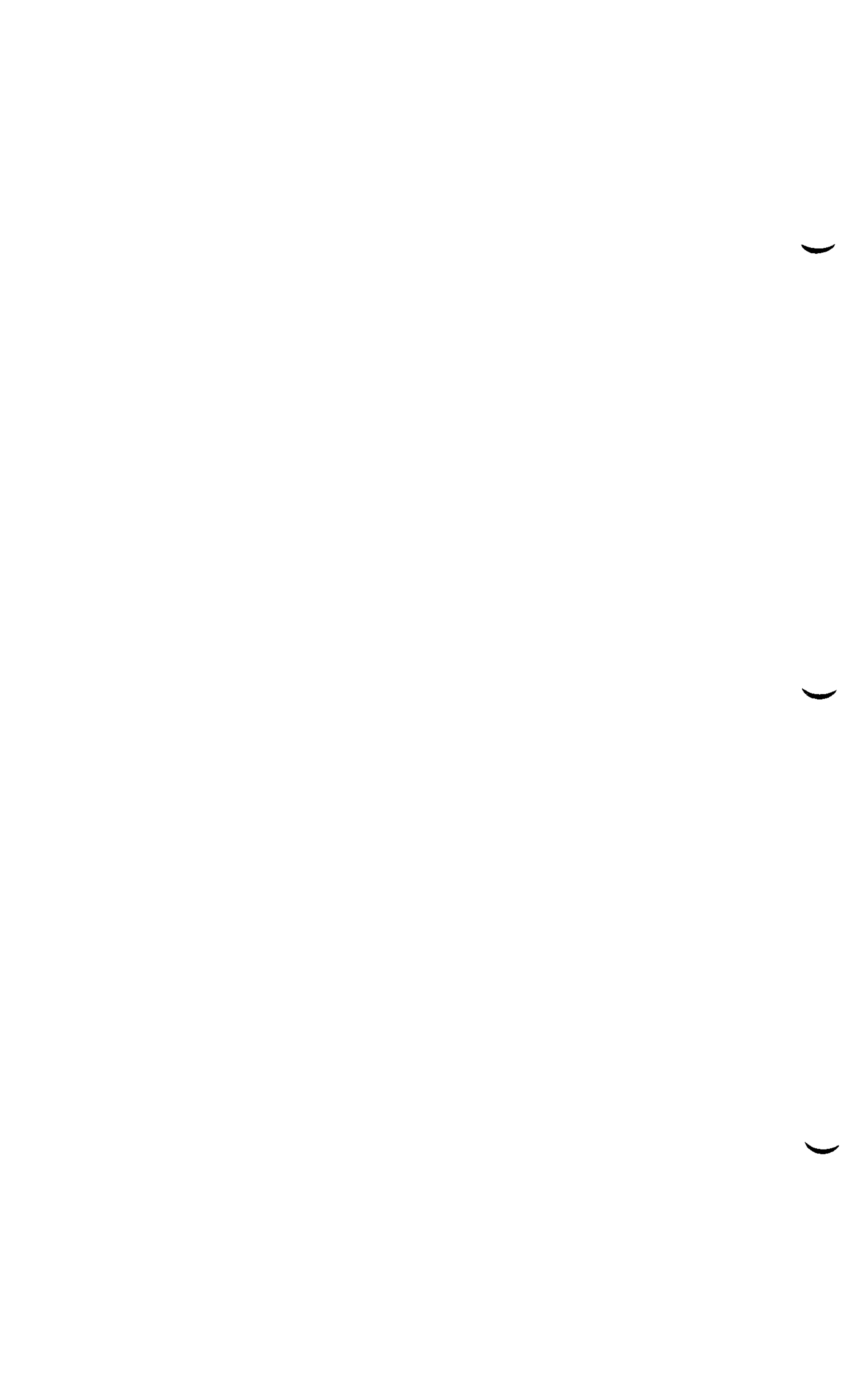
Run you program.

To run your final program, enter:

```
B: DEMO
```

Normally, the runtime system loads the BASRUNG.EXE runtime module from the default drive. Since it is not there in this example, it is looked for and found in the A: drive. Once the runtime module is loaded, execution of the file named B:DEMO.EXE begins.

This completes the demonstration run. When you exit the demonstration run, the system returns to MS-DOS. Refer to the manuals that are supplied with your MS-DOS software as well as Chapter 6, "Linking" in this manual for more information on MS-LINK.



Editing a Source Program

You need a text editor to create a GW-BASIC source program. Any text editor will do, but the most efficient choice is the full-screen editor available with NCR GW-BASIC Interpreter. If you have previous experience with GW-BASIC Interpreter, there is little need to learn how to use a new editor.

It is important to note that the compiler expects its source file in ASCII format. If you edit a file from within GW-BASIC Interpreter, it must be SAVEd with the ,A option; otherwise, the interpreter encodes the text of your program into special tokens. These tokens cannot be read by the compiler.

3.1 $INCLUDE METACOMMAND

GW-BASIC Compiler supports a useful feature that is not available when you run a GW-BASIC program under the interpreter. This is the $INCLUDE compiler metacommand. $INCLUDE is not part of an editing facility. It is a feature of the compiler that may affect the way you structure your programs. It is called a metacommand rather than a GW-BASIC command because it is not really a part of the GW-BASIC language. Rather, it is a command to the compiler, denoted by the "$" prefix. In order to avoid syntax conflicts with GW-BASIC Interpreter, always embed metacommands in comments. $INCLUDE and other metacommands are discussed in Chapter 8.

3.2 COMPILER/INTERPRETER DIFFERENCES

The interpreter supports a number of editing and file manipulation commands that are useful mainly when creating a program. LOAD, SAVE, LIST, and EDIT are examples of these commands. These operational commands are not supported by the compiler. Some differences also exist for some of the other statements and functions. Language differences must be accounted for during editing. See Chapter 4 in your *NCR GW-BASIC Reference Manual* for a full description of these differences.

3.3 LINE LENGTH

With the interpreter, maximum line length is 254 characters. (Line length is the number of characters from the beginning of the line up to the carriage return at the end. The line number is not included in the line length.) A line may be contiguous or "broken by inserting linefeeds which do not count as characters. Breaking lines is sometimes useful for readability.

With the compiler, maximum line length is 253 characters. to "break" a line, you must be using an external editor. The underscore character (_) may be used to create "logical" lines of greater than 253 characters. Using this feature, program structure and readability can be improved in some cases (the IF/THEN/ELSE statements, particularly). At the point where you want to break the line, enter an underscore as the last character before you press <RETURN> to drop down to the next line. The underscore removes the significance of the carriage return in the <RETURN> <LINEFEED> sequence that ends each line (underscore characters in quoted strings do not count in the length of characters or in significance). This results in just a linefeed being presented to the compiler. The linefeed is the line continuation character understood by the compiler and the interpreter. The ASCII key code for a linefeed is <CONTROL-J>.

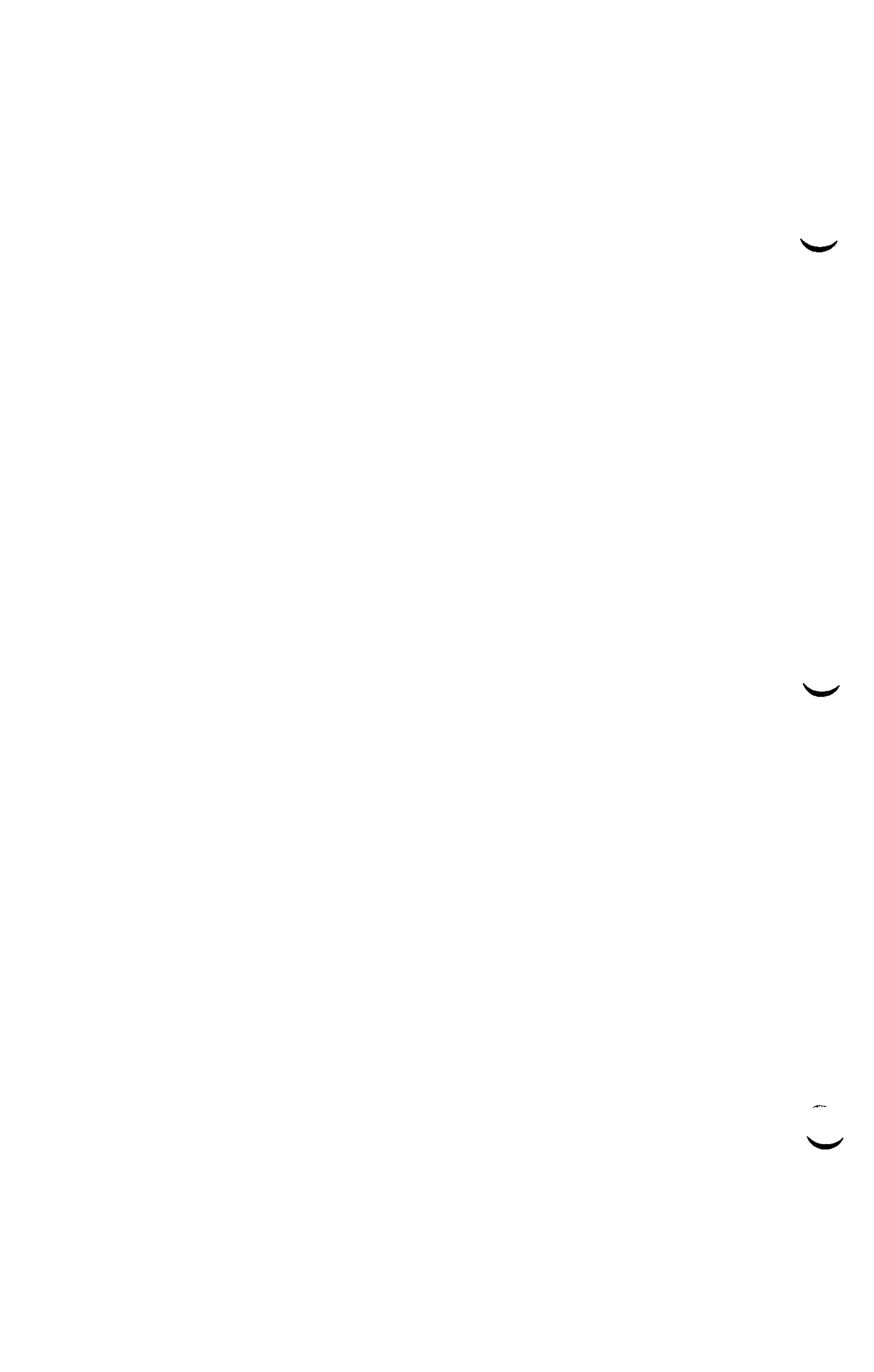
Debugging with NCR GW-BASIC Interpreter

If you have NCR GW-BASIC Interpreter, use it to debug your GW-BASIC source, i.e. to check for syntax and program logic errors. Debugging with the interpreter is an optional step. If you do not have GW-BASIC Interpreter, you must edit your program with any general purpose text editor and check for errors at compile time. We strongly urge you to complement the compiler with GW-BASIC Interpreter because the combination of the two gives you an extremely powerful and flexible GW-BASIC programming environment.

The main advantage to using the interpreter for debugging it that it stops execution of a program when an error is encountered. The program will not run and any subsequent errors are not caught until the first detected error is corrected and the program is rerun. This differs from the compiler, where all lines are scanned and all detected errors are reported at compile time. In addition, the RUN, CONT, and TRON/TROFF statements make GW-BASIC Interpreter a very powerful interactive debugging tool. See the *NCR GW-BASIC Reference Manual* for more information on these statements.

You may use some commands or functions in your compiled program that execute differently with the interpreter. In those cases, you need to use the compiler for debugging. The compiler metacommands are the only statements supported by the compiler that are not supported in some form by GW-BASIC Interpreter. In addition, the interpreter does not support double precision loop control variables and transcendental functions as does GW-BASIC Compiler.

Despite these differences, the language supported by the compiler has been made as similar as possible to GW-BASIC Interpreter. This can make GW-BASIC Interpreter your prime debugging tool, saving you debugging time by avoiding lengthy compilations and links.



Compiling

After creating a GW-BASIC source program and debugging it, your next step is to compile it. This chapter discusses GW-BASIC default file specifications, command line syntax, compiler invocation, compiler switches, and configuration.

5.1 DEFAULT FILE SPECIFICATIONS

This section describes the rules for the file specifications used when the compiler is invoked. A "default" drive or filename is the one assumed by the compiler unless otherwise specified by the user. The default file specification consists of the following:

1. A default device designation
2. A default filename extension
3. The base name of the source file parameter

The base name is the file specification less its extension and its device designation.

Table 5.1 shows how default file specifications are formed.

	Device	Extension	File Specification
Source file	dev:	.BAS	dev:base.BAS
Object file	dev:	.OBJ	dev:base.OBJ
Source listing	dev: dev:	.LST .LST	dev:NUL.LST dev:base.LST

Table 5.1 Default File Specifications

The word "base" indicates the base name.

"dev:" indicates the currently logged drive.

Capital letters are used to spell out explicit default extensions or device designations.

If the default source listing is taken, the source listing file defaults to NUL. However, if the dev: or base portion of the specification is given by the user, the default is dev:base.LST, as shown in the second example above.

Note: Specifying the NUL file is equivalent to creating no file at all. Therefore, the listing file is not created unless explicitly specified.

5.2 COMPILER INVOCATION

After you load the MS-DOS operating system, the GW-BASIC Compiler can be invoked at the command level in one of three ways:

1. Without command line options
2. With all command line options
3. With only some command line options

Each form of invocation is discussed in the following sections.

5.2.1 INVOCATION WITHOUT COMMAND LINE OPTIONS

To invoke the compiler without command line options, simply enter:

```
A:GWBCOM
```

(A: is the default drive.) The compiler then prompts for three entries. For example:

```
Source filename [.BAS]: B:MYFILE
Object filename [MYFILE.OBJ]: B:
Source listing [NULLST]: <RETURN>
```

In the previous example, the bracketed file specifications show the defaults. Explicit filename extensions or device designations override these defaults, as is shown for the object filename prompt. The device designation for the currently logged drive is the default.

By entering just a <RETURN> the user accepts the default file specification. Thus, the <RETURN> entered for the source listing file causes no file to be generated, since the default file specification is the null file (NUL.LST). (See rule 6 in the following list for more information on source listing responses.)

The rules governing input for these prompts are:

1. All lowercase letters in filenames are mapped to uppercase letters. For instance, the following three names are all considered equivalent to ABCDE.FGH:

abcde.fgh AbCdE.FgH ABCDE.fgh

2. You must enter the source filename. It has no default. Remember that the base name is the file specification less its device designation and extension.

3. To enter a file specification that contains no extension, enter the name followed by a period.

   Examples

   Source filename [.BAS]: ABC

   (ABC.BAS taken as name)

   Source filename [.BAS]: ABC.

   (ABC taken as name)

4. Entering a <RETURN> is the same as entering the default name. For example, if the currently logged drive is A:, then:

   Object filename [ABC.OBJ]: <RETURN>

   (A:ABC.OBJ taken as specification)

5. Device designations and extensions may be given to override the defaults for any prompt. For example, if the currently logged drive is A:, then:

   Object filename [ABC.OBJ]: B:

   (B:ABC.OBJ is full specification)

6. For listing files that default to null, there are two cases. In the first case, the default is specified by entering a <RETURN>. In the second case, a file is specified by entering any part of a legal file specification. In the latter case, a file is created with the same default rules that apply to other files. In particular, if a drive or extension is given, then the default base name is the base name of the source file. For example:

   Source listing [NUL.LST]: <RETURN>

   (NUL: is taken as default)

Source listing [NUL.LST]: A:

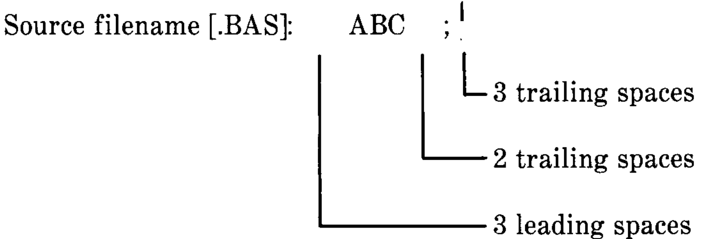
(A:ABC.LST is taken as specification, when ABC is the source filename)

7. Entry of a semicolon (;) indicates that all remaining parameters should assume their default filenames. Thus, the quick way to specify a compilation with the default options is:

   Source filename [.BAS]: ABC;

   Note, however, that a semicolon *cannot* be used to specify a default source file, since the source file has no default file specification.

8. Trailing and leading spaces are permitted. Therefore, the following is permitted:



   However, spaces *cannot* occur within filenames.

9. Switches, described in Section 5.3, "Compiler Switches," can be specified along with filenames. Switches can be placed anywhere that spaces can go.

5.2.2 INVOCATION WITH ALL COMMAND LINE OPTIONS

GW-BASIC Compiler can be invoked with command line options that make prompting unnecessary. The syntax is:

A:GWBCOM<source>,<object>,<sourcelist>

The default naming conventions that applied to the prompted responses in Section 5.2.1, "Invocation Without Command Line Options," also apply to these command line options. Options must be separated by commas. If no option is given after a comma, then the base name of the source, the default device designation, and the default extension are assumed.

For example, the invocation

A: GWBCOM DATABASE,DATABASE,DATABASE

is equivalent to:

GWBCOM DATABASE,,
Source listing [DATABASE.LST]: <RETURN>

If the normal defaults are desired with null listing files, then the semicolon (;) is used. Thus, the form

A: GWBCOM YOYO,YOYO,NUL

is equivalent to:

A: GWBCOM YOYO;

Spaces may occur before or after filenames, but not within them.

5.2.3. INVOCATION WITH SOME COMMAND LINE OPTIONS

Command line options and prompted input can be combined. This makes the compiler relatively failsafe. If options are not specified on the command line, they are requested with prompts instead.

Example

A: GWBCOM TEST,TEST
Source listing [NUL.LST]: <RETURN>

5.3 COMPILER SWITCHES

You may direct GW-BASIC Compiler to perform additional or alternate functions by adding switches to the command line.

Switches signal special instructions to be used during compilation. The switch tells the compiler to "switch on" a special function or to alter a normal compiler function. More than one switch may be used, but all must begin with a slash (/).

Examples:

A: GWBCOM DEMO/N,,NUL

A: GWBCOM DEMO,,/D
Source listing [DEMO.LST]: <RETURN>

A:GWBCOM DEMO/O
Object filename [DEMO.OBJ]: B:/X/N
Source listing [NUL.LST]: DEMO

Compiler switches fall into one of three categories:

1. Convention Switches

2. Exception Handling Switches

3. Special Code Switches

Table 5.3 summarizes the function of each compiler switch. Following the table, you will find detailed descriptions of each compiler switch category and compiler switches.

Category	Switch	Action
Convention	/4	Uses Microsoft 4.51 lexical conventions (not allowed with /N).*
	/T	Uses 4.51 execution conventions.*
Exception handling	/E	Indicates ON ERROR GOTO with RESUME <line number> in program.
	/X	Indicates ON ERROR GOTO with RESUME, RESUME 0, or RESUME NEXT in program.
	/V	Enables event trapping for communications (COM), joystick (STRIG), and function keys (KEY). Checks between lines for occurrence of an event.
	/W	Enables event trapping for communications (COM), joystick (STRIG), and function keys (KEY). Checks between statements to see if an event has occurred.
Special code	/A	Includes listing of disassembled object code in the source listing.
	/C:<size>	Allocates communication buffer size.
	/D	Generates debug code for runtime error checking.
	/N	Indicates line numbers not needed for all lines.
	/O	Substitutes the BASCOMG.LIB runtime library for BASRUNG.LIB as the default runtime library searched by the linker.
	/R	Stores arrays in row major order (as does MS-Pascal).
	/S	Writes quoted strings to OBJ file on disk and not to compiler symbol table in memory.

*Use /4 and /T together for 4.51 lexical and execution conventions. Do not use /4 and /N together.

Table 5.3 Compiler Switches

5.3.1 CONVENTION SWITCHES

The convention switches specify use of MS-BASIC Version 4.51 lexical (language) and execution conventions during compilation. If these switches are not specified, MS-BASIC Version 5.0 conventions are used.

Switch	Action
/4	The /4 switch directs the compiler to use the lexical conventions of MS-BASIC Interpreter version 4.51. Lexical conventions are the rules that the compiler uses to recognize the MS-BASIC language. The following lexical conventions are observed with /4: 1. Spaces are not significant. 2. Variables with embedded reserved words are illegal. 3. Variable names are restricted to two significant characters. The /4 switch is needed to correctly compile a source program in which spaces do not delimit reserved words, as in the following statement: FORI=ATOBSTEPC Without the /4 switch, the compiler would assign the variable "ATOBSTEPC" to the variable "FORI". With the /4 switch set, the compiler recognizes the line as a FOR statement. NOTE: The /4 and /N switches *may not* be used together.
/T	The /T switch tells the compiler to use MS-BASIC version 4.51 execution conventions. Execution conventions govern the implementation of MS-BASIC functions and commands and what they actually do at runtime. With the /T switch specified, the following 4.51 execution conventions are switched on: 1. FOR/NEXT loops are always executed at least one time.

2. TAB, SPC, POS, and LPOS functions perform according to 4.51 conventions. For the other functions, no convention differences exist between versions 4.51 and 5.0.

3. Automatic floating-point to integer conversions truncate numbers rather than rounding them, except in the case where a floating-point number is being converted to an integer in an INPUT statement.

4. The INPUT statement does not affect the variables in the input list if only a carriage return is entered. If a "?Redo from start" message is issued, then a valid input list must be given. A carriage return in this case generates another "?Redo from start" message.

   Note that a "?Redo from start" message can only be generated following an INPUT statement; it is not a standard GW-BASIC error message.

5.3.2 EXCEPTION HANDLING SWITCHES

The /E and /X switches are error handling switches that allow use of ON ERROR GOTO and RESUME statements in your program. The /V and /W switches are event trapping switches that enable the event trapping facility and check between lines or statements to see whether an event has occurred. Note that these switches add extra code to your program and cause more time to be used for compilation.

/E — The /E switch tells the compiler that the program contains an ON ERROR GOTO/RESUME <line number> construction. To handle ON ERROR GOTO statements properly, the compiler must generate extra code for the GOSUB and RETURN statements. Also, a line number address table (one entry per line number) must be included in the binary file, so that each runtime error message can include the number of the line in which the error occurs.

This switch should not be used unless the program contains an ON ERROR GOTO statement.

NOTE: If a RESUME statement other than RESUME <line number> is used with the

ON ERROR GOTO statement, the /X switch should be used, rather than /E.

/X — The /X switch tells GW-BASIC Compiler that the program contains one or more RESUME, RESUME NEXT, or RESUME 0 statements. The /X switch performs all the functions of the /E switch, so the two need never be used at the same time. For instance, the /X switch, like the /E switch, causes a line number address table to be included in the binary object file, so that each runtime error message can include the number of the line in which the error occurs. Note, however, that /X provides one table entry per statement, whereas /E provides one entry per line number.

In order that RESUME statements may be handled properly, the compiler cannot optimize across statements. Therefore, /X should not be used unless the program contains RESUME statements *other than* RESUME <line number>.

/V — The /V switch enables event trapping for communications (COM), joystick (STRIG), and function keys (KEY). It also checks between every line in the program to see whether an event has occurred.

/W — Like the /V switch, /W enable event trapping, but it checks between every *statement*, including multiple statements on a line, to see whether an event has occurred.

5.3.3 SPECIAL CODE SWITCHES

The special code switches enable particular compiler options.

/A — The /A switch generates a listing of the disassembled object code for each source line, and shows precisely the code that is being generated by the compiler. This switch can greatly increase the length of a listing. It has no effect on the actual code generated by the compiler.

/C:<size> — The /C:<size> switch will allocate the <size> for the communication buffer of a GW-BASIC program. The default is 256 bytes if /C:<size> is not used.

/D

The /D switch causes debugging and error handling code to be generated at runtime. Use of /D allows use of TRON and TROFF in the compiled file. Without /D set, TRON and TROFF are ignored.

With /D, GW-BASIC Compiler generates somewhat larger and slower code that checks the following:

1. Arithmetic overflow

   All arithmetic operations, both integer and floating-point, are checked for overflow and underflow.

2. Array bounds

   All array references are checked to see if the subscripts are within the bounds specified in the DIM statements.

3. Line numbers

   The generated binary code includes line numbers so that the runtime error listing can indicate on which line each error occurs.

4. RETURN

   Each RETURN statement is checked for a prior GOSUB statement.

Without the /D switch set, array bound errors, RETURN without GOSUB errors, and arithmetic overflow errors do not generate error messages at compile time or runtime. The result may be erroneous program execution.

/N

The /N switch relaxes line numbering constraints. When /N is specified, line numbers in the source file may occur in any order, or they may be eliminated entirely.

With /N, lines are compiled normally, but unnumbered lines cannot be targets for GOTO or GOSUB statements. While /N is set, the underline character causes the remainder to the physical line to be ignored. Also, /N causes the underline character to act as a linefeed so that the next physical line becomes a continuation of the current logical line.

There are three advantages to using the /N switch:

1. Elimination of line numbers increases program readability.

2. GW-BASIC Compiler optimizes over entire blocks of code rather than single lines (for example in FOR/NEXT loops.)

3. GW-BASIC source code may more easily be included in a file with the $INCLUDE metacommand.

NOTE: /N should not be used with /4.

/O

The /O switch tells the compiler to substitute the BASCOMG.LIB runtime library for BASRUNG.LIB as the default runtime library searched by the linker. This switch cannot be used with the runtime module. Any EXE files created by linking to BASCOMG.LIB do not need the runtime module on disk at runtime.

/R

The compiler normally stores arrays in column major order. The /R switch instructs the compiler to store arrays in row major order. This permits languages, such as Microsoft Pascal Compiler, which normally store arrays in row major order, to access such arrays. Note that the GW-BASIC Interpreter and Microsoft FORTRAN Compiler store and access arrays in *column* major order.

/S

The /S switch forces the compiler to write quoted strings that exceed four characters to an OBJ file on disk as they are encountered, rather than retaining them in memory during the compilation of the program. If this switch is not set, and the program contains a large number of long quoted strings, the user may run out of memory at compile time.

Although the /S switch reduces the amount of memory used at compile time, it may *increase* the amount of memory needed in the runtime environment, since multiple instances of identical strings will exist in the program. Without /S, references to multiple identical strings are combined so that only

one instance of the string is necessary in the final compiled program.

5.4 CONFIGURING FOR GW-BASIC

Before you execute a compiled GW-BASIC program, you may need to first run the GW-CONF routine. GWCONF, which is on the GW-BASIC Compiler disk, is used to define configuration information to GW-BASIC. Specifically, you must run GWCONF if any of the following conditions apply to your compiled GW-BASIC program:

- You are using a printer to print graphics.
- You want to print either red or blue memory.
- You are using communications.

You must define your printer even if you've already defined it with the MS-DOS Configure routine: GW-BASIC requires additional information.

The following table summarizes what information you can specify with GWCONF. Note that if you select a printer with a serial interface, certain definitions are automatically set up.

	Initial Definition	With GWCONF
Printer	None	EPSON FX80 ITOH M8510A (bi-directional) ITOH M8510A (one-directional) None
Serial Printer Interface: — Stop Bits — Parity — Character length — Baud rate	 1 even 7 bits 9600	 1 1/2 or 2 disabled or odd 5, 6, or 8 50 - 19200
Memory Printout	Green	Red or blue
Communication Port Addresses	Port 1 = 00 hex Port 2 = 00 hex	30, 38, 60, 68, 70, 78, B0, B8, C0, C8.

To use GWCONF, insert the GW-BASIC Compiler disk. (MS-DOS must already be loaded.)

NOTE: The GW-BASIC Compiler disk may be inserted either in drive A or drive B. On single flexible disk drive systems, disk "swapping" must be performed.

Type GWCONF; you see the following screen:

```
1) Select Printer
2) Modify Color for Screendump
3) Modify Communication Ports
4) Exit Program

 * Enter Your Selection
```

For each entry, further screens guide you through your definition. Although the screens are self-explanatory, some usage conventions may be helpful. (This description, however, assumes you know the communications port addresses for your configuration. This information is detailed in the "System Technical Manual, Part 1, Hardware.")

You can select any of the three memories for your screen dump, but you can only print one at a time; therefore, if you want to print all three images, you must use GWCONF before each print run.

If you have a monochrome machine, you must select green foreground color dump before executing your program.

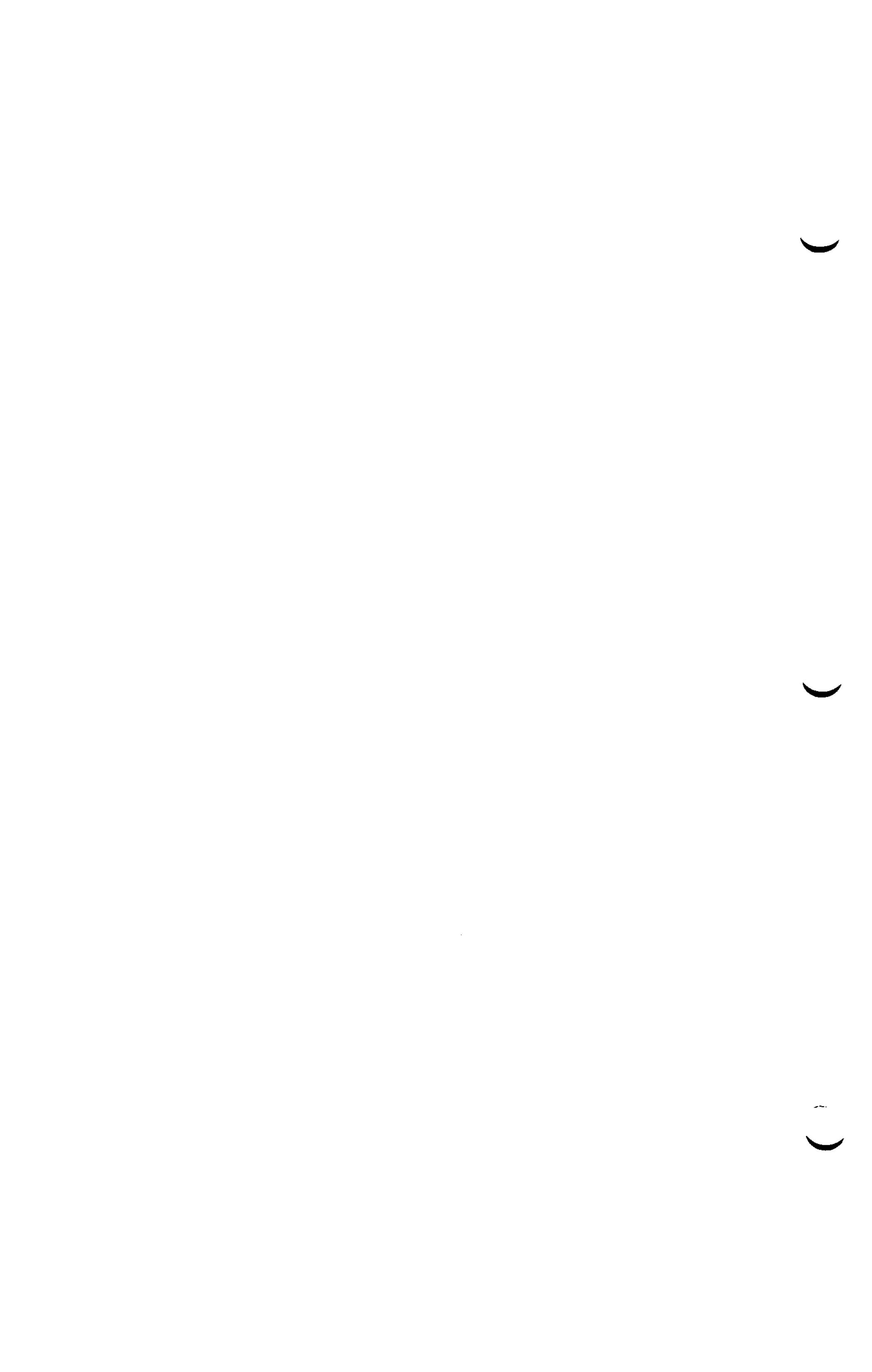
The Exit Program function is used after you complete your modifications. When specified, the software displays:

```
1) Update O.S. disk in drive A
2) Return to main program
3) Exit CONFIG

 * Enter Function
```

Function 1 is used to have the new configuration information written to MS-DOS master disk; be sure the MS-DOS is in drive A. If the modifications are only temporary (for the current run), use function 3; the changes are only made in memory.

After you run GWCONF, you are ready to execute your GW-BASIC program.



Linking

Compiled object files must be linked to one of the runtime libraries before they can be executed. This section explains the differences between the two runtime libraries available with GW-BASIC. For discussion of how to use the linker, see your MS-DOS documentation.

There are two ways to link an object file:

1. Link to the BASRUNG.LIB runtime library.

2. Link to the BASCOMG.LIB runtime library.

In the first case, the runtime module (BASRUNG.EXE) is used at runtime. The runtime module contains the routines most commonly used during runtime. In the second case, selected routines are linked to the OBJ file to create a single EXE file that does not need the runtime module.

The default and preferred method is to link to the BASRUNG.LIB runtime library. In some cases (described below), the second method may be preferable. In either case, the goal is to produce an executable EXE file by processing a compiler-produced OBJ file with MS-LINK.

6.1 LINKING TO BASRUNG.LIB RUNTIME LIBRARY

MS-LINK links programs, assembly language routines, and library routines. This allows you to incrementally develop large programs by separately compiling or assembling parts of a program and then linking those parts together.

For example, assume that you have created a GW-BASIC program named PROG.BAS that uses two external assembly language procedures, ASM1.ASM and ASM2.ASM. Assume also that the program has already been compiled, and that the assembly language routines have already been assembled. The files created from the compilation and assemblies are:

```
PROG.OBJ
ASM1.OBJ
ASM2.OBJ
```

To link these all together, first invoke MS-LINK:

```
A: LINK
```

MS-LINK prompts you for the names of the .OBJ files that you want to link together. At this point, respond to the prompts as follows:

```
Object Modules [.OBJ]: PROG+ASM1+ASM2
Run File [A:PROG.EXE]:;
```

MS-LINK automatically links your .OBJ file to routines in BASRUNG.LIB. (If you want to link to BASCOMG.LIB, you must use the /0 compiler switch. This is described in Section 6.2, "Linking to BASCOMG.LIB Runtime Library.") Not using the /0 switch means that MS-LINK searches the runtime library for the routines that are to be linked to the final load module.

In very rare cases, you might want to explicitly specify a search of BASRUNG.LIB. If you wish to link object modules without any library modules (which is also a very rare circumstance), you must give MS-LINK an empty library to search. Refer to the manuals that are supplied with your MS-DOS software if you need additional information on these special applications.

Linking with the BASRUNG.LIB runtime library provides the following advantages:

1. COMMON and CHAIN statements can be used to support a system of programs sharing common data. With BASCOMG.LIB, COMMON is not supported, and CHAIN is the semantic equivalent of RUN.
2. The BASRUNG.EXE runtime module resides in memory, and therefore does not need to be reloaded for each program in a system of chained programs.
3. The routines in BASRUNG.EXE are not incorporated into your EXE file. Therefore, for a system of several EXE files on a disk, considerably less disk space will be required than with BASCOMG.LIB.
4. Code generated by linking with BASRUNG.LIB can be as much as 15 to 20 percent shorter than the code generated by linking with BASCOMG.LIB.

6.2 LINKING TO BASCOMG.LIB RUNTIME LIBRARY

When BASCOMG.LIB is selected as the library to be searched, the program does not use the runtime module, BASRUNG.EXE.

This is because linking with BASCOMG.LIB produces a single EXE file that already contains the library routines needed for execution.

If the program is to be linked to the BASCOMG.LIB library, the /0 switch must be specified at compile time. When /0 is specified, the alternate runtime library (BASCOMG.LIB) is substituted for BASRUNG.LIB as the default library to be searched at link time.

In the following cases, it might be advantageous to link with BASCOMG.LIB rather than with BASRUNG.LIB:

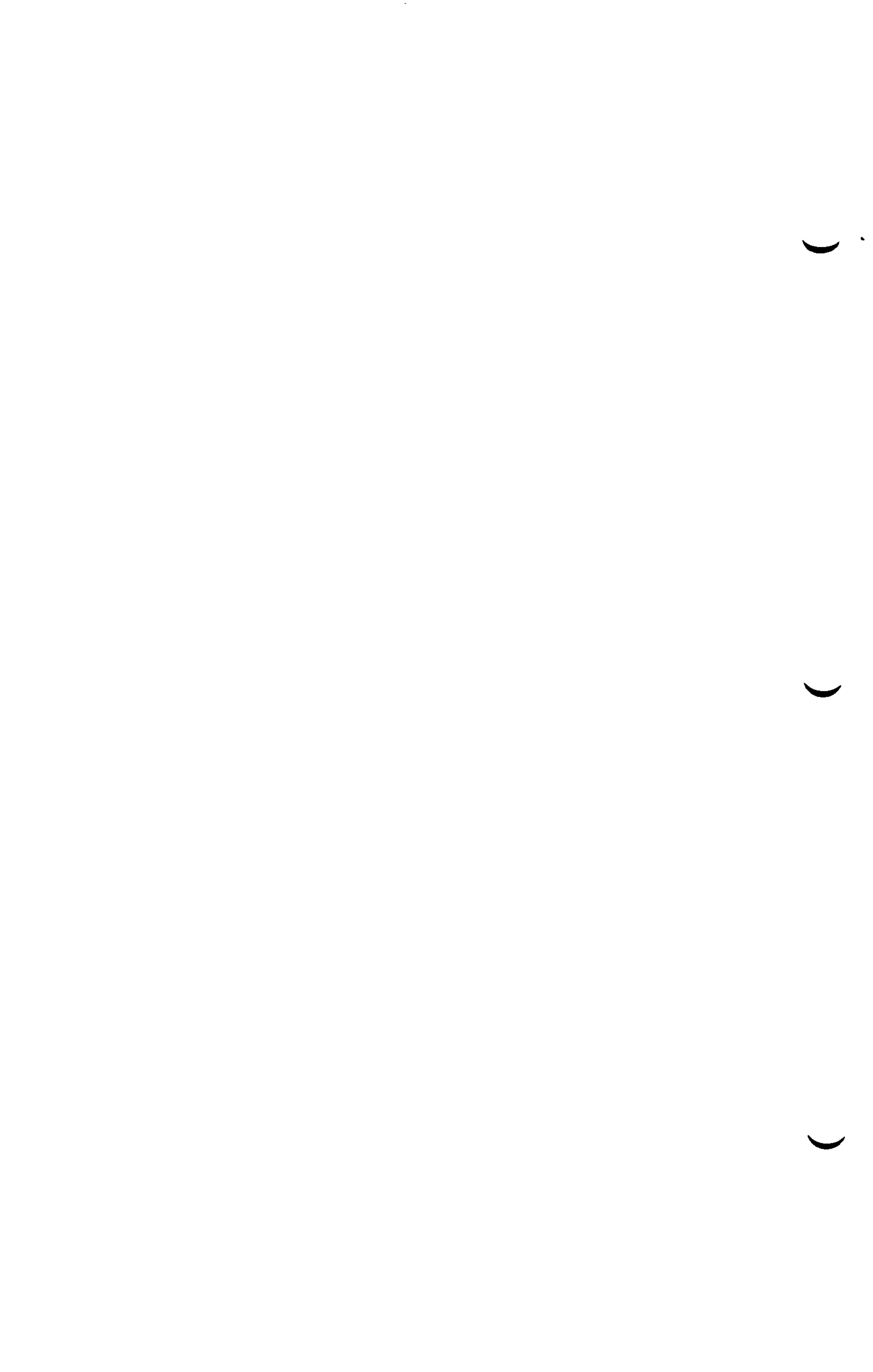
1. For small, simple programs that do not require all the routines in the runtime module, you may save space by linking with BASCOMG.LIB.
2. With BASCOMG.LIB, execution of a compiled and linked EXE file does not require that the runtime module be on disk at runtime.
3. With BASCOMG.LIB, programs execute slightly faster than programs linked with BASRUNG.EXE because the runtime routines are invoked through 8086 intersegment calls. With BASRUNG.LIB, the runtime routines are invoked through software interrupts.

Three precautions should be taken when linking GW-BASIC programs:

1. The /DSALLOCATION switch should always be set (this is the default).

2. Your programs should always be loaded low (i.e., the /HIGH switch should not be used.)

3. The name of the GW-BASIC program should always be given as the *first* .OBJ file to be loaded; otherwise (e.g., if an assembly language module is loaded first), segments may be ordered incorrectly.



Running a Program

To run a program that has been compiled and linked, enter the filename without its .EXE filename extension. For example:

B: DEMO

This command executes the program DEMO.EXE. If the program DEMO.EXE was linked to BASRUNG.LIB, the BASRUNG.EXE runtime module is loaded from the default drive (drive B: in this example). If BASRUNG.EXE is not on the default drive, then the runtime system looks for it in the A: drive. If it is still not found, the following message is displayed:

Cannot find A:BASRUNG.EXE
Enter new drive letter:

At this point, enter the drive where the file is located, followed by a <RETURN>. Once the runtime module is loaded, execution of the file named B:DEMO.EXE begins.

The executable binary file can also be executed from within a program, as in the following statement:

10 RUN "PROG"

The default extension is .EXE. Note that an .EXE file can be a binary file created in any programming language. The CHAIN statement is used similarly. In either case, an executable binary file is loaded. The runtime module is not reloaded when you use CHAIN; it is when you use RUN.

The bulk of the runtime environment is taken up by the runtime module. This module is automatically loaded when you initially invoke an .EXE file requiring the runtime module. When you RUN a program, the .EXE file is loaded into memory. Both the program and the runtime module files reside in memory simultaneously. See Appendix C, "Memory Maps," for a diagram of a runtime memory map.

Metacommands

Metacommands are compiler directives that control source files and listing files. The available metacommands are listed in Table 8, below, and are described in the text following the table.

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>$INCLUDE:'<filename>'</td><td>Switches compilation from current source file to source file given by <filename>.</td></tr>
<tr><td>$LIST{+|-}</td><td>Turns on/off source file listing.
Errors are always listed.</td></tr>
<tr><td>$'OCODE{+|-}</td><td>Turns on or off disassembled object code listing.</td></tr>
<tr><td>$TITLE:'<text>'</td><td>Sets page title.</td></tr>
<tr><td>$SUBTITLE:'<text>'</td><td>Sets page subtitle.</td></tr>
<tr><td>$LINESIZE:n</td><td>Sets width of listing.
Default is 80 characters.</td></tr>
<tr><td>$PAGESIZE:n</td><td>Sets length of listing in lines.
Default is 66 ; 60 are printable.</td></tr>
<tr><td>$PAGE</td><td>Skips to next page. Line number is reset.</td></tr>
<tr><td>$PAGEIF:n</td><td>Skips to next page if less than (n) lines left.</td></tr>
<tr><td>$SKIP:n</td><td>Skips (n) lines or to end of page.</td></tr>
</table>

Table 8 The Metacommands

8.1 SYNTAX

One or more metacommands can be given at the start of a comment. Multiple metacommands are separated by the whitespace characters space, tab, or linefeed. Whitespace between parts of a metacommand is ignored. Therefore, the following metacommands are equivalent:

```
REM $PAGE:12
REM $PAGE : 12
```

Note, however, that no space may appear between the dollar sign and the rest of the metacommand.

Except for $INCLUDE, the metacommands affect the source listing only. Many metacommands can be turned on and off within a listing. For example, most of a program might use $OCODE-, with a few sections using $OCODE+ as needed. However, some metacommands, due to their nature, apply to an entire compilation.

The following rules apply to the use of metacommands:

1. A metacommand followed by a plus sign (+) or minus sign (-) is an on/off switch.

2. A metacommand followed by :n requires an integer (0 < n > 256).

3. A metacommand followed by :'<text>' requires a string of characters enclosed in single quotation marks.

8.2 DESCRIPTIONS

$INCLUDE:'<filename>'

The $INCLUDE:'<filename>' metacommand tells the compiler to switch processing from the current source file to the GW-BASIC file given by the <filename> parameter. When the end-of-file is reached in the included source file, the compiler switches back to the original source file and continues compilation. Resumption of compilation in the original source file begins with the line of text that follows the line in which the $INCLUDE occurred. Therefore, REM $INCLUDE should always be the last statement on its line, since the remainder of the line is always treated as part of a comment.

Included files may be subroutines, single lines, or partial programs. <filename> must be surrounded by single quotation marks. The default extension is BAS.

Take care that any variables in the included files match their counterparts in the main program, and that included lines do not contain erroneous code such as GOTO statements to nonexistent lines or END statements.

These further restrictions must be observed:

1. Included files must be in ASCII format; i.e., they must have been saved with the ,A option if they were created from within the GW-BASIC Interpreter.

2. Included lines must be numbered in ascending order.

3. The lowest line number of the included lines must be higher than the line number of the $INCLUDE metacommand in the main program.

4. The range of line numbers in the included file must numerically precede subsequent line numbers in the main program. This and the two previous restrictions are removed if the main program is compiled with the /N switch set, since line numbers need not be in ascending order in this case.

5. $INCLUDE metacommands can be nested to five levels, counting the main source program.

6. The $INCLUDE metacommand must be the last statement on a line, and must be part of a comment statement, as in the following statement:

   999 DEFINT I-N : REM $INCLUDE: 'COMMON.BAS'

All other metacommands are designed to control the source file listing (see below). Note, however, that none of the remaining metacommands have any effect if NUL.LST is the name of the source listing file.

$LIST{+ | -}
The $LIST+ metacommand turns on the source listing; $LIST turns it off. Metacommands themselves appear in the listing, except for $LIST-.

$OCODE{+ | -}
Controls listing of the generated code in the listing file.

For each GW-BASIC source line, code addresses and operation mnemonics are listed. $OCODE- turns off listing of the generated code, even if the /A switch is used when the compiler is invoked. $OCODE+ turns on the generated code listing, regardless of the use of /A.

$TITLE:'<text>'

Prints the title specified by <text> at the top of each page of the source file listing. The string <text> must not exceed 59 characters.

$SUBTITLE:'<text>'

Prints the subtitle specified by <text> beneath the title at the top of each page of the source file listing. The string <text> must not exceed 59 characters.

$LINESIZE:n

Sets the maximum length of lines in the source listing file. The default length is 80 characters. The number of characters printed per line is (n - 1). The integer n must be greater than 40.

$PAGESIZE:n

Sets the maximum size of a page in the source file listing. The default size is 66 lines. In order to allow space for the page header, a page has (n-6) lines printed on it. The integer n must be 15 or greater.

$PAGE

Forces a new page in the source file listing. The page number of the listing file is automatically incremented.

$PAGEIF:n

Conditionally performs $PAGE, above, if there are fewer than n printed lines left on the page. If there are n or more lines left on the page, no action is taken.

$SKIP:n

Skips n lines in the source listing file. If there are fewer than n lines left on the current page, the listing skips to the start of the next page.

A Compiler/Interpreter Comparison

The differences between the languages supported by the GW-BASIC Compiler and the GW-BASIC Interpreter fall into three categories: operational differences, implementation differences, and language differences. This section describes operational and implementation differences. Compiler metacommands are described in Chapter 8. Other language differences (commands, statements, and functions) are described in the *NCR GW-BASIC Reference Manual.*

9.1 OPERATIONAL DIFFERENCES

Certain commands are designed for the interactive programming environment of GW-BASIC Interpreter and are not used with the GW- BASIC Compiler. These commands are:

AUTO
CONT
DELETE
EDIT
LIST
LLIST
LOAD
MERGE
NEW
RENUM
SAVE

Other commands, however, are used with the compiler but not with the interpreter. These commands are called "metacommands."

9.2 IMPLEMENTATION DIFFERENCES

Implementation differences include:

1. Floating-Point Calculations

   Numeric calculations involving numbers with a large number of decimal places may not exactly produce the same results as the

same calculations performed with the interpreter. This difference affects only calculations involving very precise numbers.

2. Expression Evaluation

During expression evaluation, the GW-BASIC Compiler converts operands of different types to the type of the more precise operand.

For instance, the following expression causes J% to be converted to single precision and added to A!:

```
QR=J%+A!+Q#
```

The resultant sum is then converted to double precision and added to Q#.

Note that the GW-BASIC Compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter, the following statements yield 40000 for A%:

```
I%=20000
J%=20000
A%=I%+J%
```

That is, J% is added to I%. Because the number is too large for an integer representation, the interpreter converts the result into a floating-point number. The result (40000) is found and converted back to an integer and saved as A%.

The GW-BASIC Compiler, however, must make type conversion decisions during compilation. It cannot defer until actual values are known. Thus, the compiler generates code to perform the entire operation in integer mode and arithmetic overflow occurs. If the /D (Debug) switch is set, the error is detected. Otherwise, an incorrect answer is produced.

When the above example is executed with the compiler, the I% +J% yields the integer value -25536, which is then converted to a floating-point value and saved in A%.

Besides these type conversion decisions, the compiler performs certain valid optimizing algebraic transformations before generating code. For example, the following program could produce an incorrect result when run:

```
I% =20000
J% =-18000
K% =20000
M% =I% +J% +K%
```

If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I% +K% first and then adds J%, overflow does occur. The compiler follows the rules of operator precedence. But no other guarantee of evaluation order can be made; even the use of parentheses may not always direct the order of evaluation.

3. Integer Variables

   To produce the fastest and most compact object code possible, use integer variables whenever possible. For example, the following program executes approximately 30 times faster when the loop control variable "I" is replaced with "I%", or when I is declared an integer variable with DEFINT.

```
FOR I=1 TO 10
A(I)=0
NEXT I
```

   It is especially advantageous to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

4. Double Precision Arithmetic Functions

   The GW-BASIC Compiler allows use of double precision floating-point numbers as operands for arithmetic functions, including all of the transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, and SQR). Only single precision arithmetic functions are supported by the interpreter.

5. Double Precision Loop Control Variables

   The compiler, unlike the interpreter, allows the use of double precision loop control variables. This lets you increase the precision of the increment of increase the range of loops.

6. String Size

   The compiler supports strings of up to 32767 characters. To support such an implementation, each string descriptor requires 4 bytes of memory.

7. String Space Implementation

The compiler and interpreter differ in their implementation and maintenance of string space. Using either POKE with PEEK and VARPTR, or using assembly language routines to change string descriptors may cause a "String Space Corrupt" error message.

Communications

This chapter describes the BASIC statements required to support RS-232 asynchronous communication with other computers and peripherals (with or without XON-XOFF Protocol). Note that the NCR GW-BASIC Compiler also supports K211 and K215 asynchronous interfaces.

10.1 OPENING A COMMUNICATIONS FILE

The OPEN COM statement allocates a buffer for input/output in the same manner as the OPEN statement for disk files. Refer to the OPEN COM statement in Chapter 4 of the *NCR GW-BASIC Reference Manual.*

10.2 COMMUNICATION I/O

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

Communications sequential input statements are the same as those for disk files. They are:

INPUT#
LINE INPUT#
INPUT$

Communications sequential output statements are also the same as those for disk files. They are:

PRINT
PRINT USING
WRITE#

Refer to your *NCR GW-BASIC Reference Manual* for details on format and usage of the above statements and functions.

10.2.1 I/O FUNCTIONS

The most difficult aspect of asynchronous communication is processing characters as fast as they are received. At rates above 2400 bps it is necessary to suspend character transmission from the host long enough for characters already received to be processed. This can be done by sending XOFF (CONTROL-S and XON (CONTROL-Q) to the host computer. XOFF tells the host to stop sending, and XON tells it to resume sending.

There are three functions which help to determine when an overrun condition may occur:

LOC(x)	Returns the number of characters in the input buffer which are waiting to be read. If more than 255 characters are in the buffer, LOC(x) returns 255. (The input buffer can hold more than 255 characters, as determined by the /C: option on the BASIC command.) If fewer than 255 characters remain in the buffer, LOC(x) returns the actual amount.
LOF(x)	Returns the amount of free space in the input buffer. This is the same as /C:<size>-LOC(x), where size is the size of the communications buffer as set by the /C: option. The default size of the buffer is 256.
EOF(x)	Returns true (-1) if the input buffer is empty; returns false (0) if there are any characters waiting to be read.

10.2.2 INPUT$ FUNCTION

As a recommendation, use the INPUT$ function instead of the INPUT# and LINE INPUT# statements when reading communications files, because it allows all characters read to be assigned to a string. INPUT# stops input when it detects a comma or carriage return.

INPUT# returns a string of x characters read from the file number Y. The following statements are efficient in reading a communications buffer:

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

If there are characters in the input buffer, the above statements return the characters in the buffer into A$ and process them (lines 30, 40, 50, etc.). If there are more than 255 characters, only 255 at a time will be returned to prevent string overflow. Further, if there are more than 255 characters, EOF(1) is false, and input into A$ continues until the buffer is empty.

10.2.3 GET AND PUT STATEMENTS FOR COMMUNICATIONS

GET and PUT are only slightly different for communications files than for disk files.

Syntax

GET <file number>,<nbytes>
PUT <file number>,<nbytes>

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

nbytes
Specifies number of bytes to be transferred into or out of the communications file.

Purpose

Allows for fixed length I/O to or from the communications file.

Remarks

Because of the low performance associated with telephone line communication, it is recommended that GET and PUT not be used in such applications.

10.3 CONTROL SIGNALS

This section contains information about control signals which you may need to know in order to communicate with another computer or peripheral.

10.3.1 OUTPUT SIGNALS

When you start BASIC on the NCR Decision Mate V, the Request to Send (RTS) and Data Terminal Ready (DTR) signal lines are not turned on until an OPEN COM statement is performed. You can suppress the RTS signal by specifying the RS option in the OPEN COM statement (refer to Chapter 4 of your *NCR GW-BASIC Reference Manual).* Unless suppressed, the line stays on until the communications file is closed by CLOSE, END, NEW, RESET, SYSTEM, or RUN without the R option. If an OPEN COM statement fails, the lines remain on. You may then retry the OPEN COM statement without using a CLOSE statement.

10.3.2 INPUT SIGNALS

If either the Clear To Send (CTS) or Data Set Ready (DSR) signal lines are off, you cannot run an OPEN COM statement. BASIC returns a "Device Timeout" error after one second. You can, however, specify if and how you want these lines tested by using the CS and DS options in the OPEN COM statement.

If the CTS or DSR line signals are off while a program is running, I/O statements associated with the communications file will not work, and a "Device Fault" or "Device Timeout" error occurs.

If the host computer is running a program and the satellite computer sends characters, only one character will be saved in the host's hardware interface. Then the next time a communications statement is run by the host computer, a "Device I/O" error occurs. This indicates an overrun on the host's hardware interface.

10.4 SAMPLE PROGRAM

The following program enables the NCR DECISION MATE V to be used as a conventional terminal. In addition to full-duplex communication with a host, the program allows data to be downloaded (written) to a file, and conversely, a file may be up-loaded (transmitted) to another machine.

In addition to demonstrating the elements of asynchronous communications, this program should be useful in transferring BASIC programs and data to and from the NCR DECISION MATE V.

Notes on the Sample Program

Line No.	Comments
	When starting GW-BASIC, set the /F: switch to 3.

10	Sets the screen to normal alpha mode.
20	Turns off the programmable function key display, clears the screen, and makes sure that all files are closed.

NOTE: Asynchronous implies character I/O as opposed to line or block I/O. Therefore, all PRINTs (either to the communications file, the screen, or a disk file) are terminated with a semicolon (;). This stops the carriage return normally issued at the end of a PRINT statement.

30	Defines all numeric variables as integers. This is primarily for use in the subroutine at lines 500-660. Any program looking for speed optimization should use integer counters in loops wherever possible.
35-40	Clears the 23rd line starting at column 1.
50	Defines Boolean true and false.
70	Defines the ASCII XON and XOFF characters.
100-130	Prints program identification and asks for baud rate (speed). Opens communications to file number 1 with even parity, 7 data bits, and a line feed (LF) following every carriage return.
200-280	This section gives you a menu for receiving data at your screen or on a file, or for transmitting data from your keyboard or from one of your files.

1. You are asked how many characters have to be received on your communication line before they are displayed on the screen.

2. Reads one or more characters from the keyboard into A$ and transmits A$. You are guided by the menu to continue.

3. If only a space is entered, wait for n characters and print them when received.

4. If the character was M only, then the user is ready to down-load a file, so get file name.

5. If you entered an E only, the program will stop at 9000-9040.

6. If the input (A$) is not M, E, or space, send it by writing to the communication file (PRINT #1...), as described in step 2, and at line 230 go back to menu.

7. At lines 250-260, read and display contents of communications buffer (as much as selected by n) on screen. Continue with 1.

300-310	Get disk file name to be used
400-430	Asks if file name is to be transmitted (up-loaded) or received (down-loaded) and opens file.
490-540	The received data will fill an array of 126 positions unless an end-of-file character (line 530) was received, which closes the file.
550-620	Before writing to the selected disk file, an XOFF is sent to the transmitter. Two additional characters (lines 560-590) may be read after the 126 positions are filled and before the transmitter gets the XOFF.
625	When the array is completely written to disk file and XON is sent to the transmitter, the transmitter continues sending.
630	Continue receiving as at line 500.
640-680	For end-of-file, write last characters to file and close it. Continue again at the menu.
800-880	This is a waiting routine used when the transmitter also receives characters. If the transmitter receives an XOFF, wait until XON is received before continuing transmission.

1000-1060	This is a transmit routine. Until the end of the disk file: Read one character into A$ with INPUT$ statement. Send character to communications device in 1015. (If a character is received, the waiting routine for XON in case of XOFF is called, line 1015.) Send a CONTROL-Z at the end-of-file in line 1040 in case the receiving device needs one to close its file. Finally, in lines 1050 and 1060, close disk file, print completion message, and go back to conversation mode in line 200.
9000-9040	These lines are run if you enter E in response to your menu. These lines close the communications file and the screen output file, restore the programmable function key display, and end the program.

```
10 SCREEN 0
20 KEY OFF:CLS:CLOSE
30 DEFINT A-Z
35 LOCATE 23,1
40 PRINT STRING$(60," ")
50 FALSE=0:TRUE= NOT FALSE
70 XOFF$=CHR$(19):XON$=CHR$(17)
100 LOCATE 23,1:PRINT "Async TTY Program        ";
110 LOCATE 1,1:LINE INPUT "speed? ";SPEED$
120 COMFIL$="coml:"+SPEED$+",e,7,,LF
130 OPEN COMFIL$ AS #1
140 OPEN "scrn:" FOR OUTPUT AS #2
200 LOCATE 1,1:LINE INPUT "on receiving, wait for n char, n=";N$
203 N%=VAL(N$)
205 LOCATE 3,1:PRINT "press any keys for transmission"
206 PRINT "except: M   for file i/o"
207 PRINT "or      space for receiving"
208 PRINT "or      E     for ending program"
209 LINE INPUT:A$
210 If A$=" "THEN 250
211 IF A$="M" THEN 300
212 IF A$="E" THEN 9000
220 PRINT #1.A$;
230 GOTO 200
250 A$=INPUT$(N%,#1)
260 PRINT #2,A$;
280 GOTO 200
300 LOCATE 8.1
310 LINE INPUT"file? ";DSKFIL$
400 LOCATE 9.1
410 LINE INPUT"(T)ransmit or(Receive? ";TXRX$
420 IF TXRX$="T" THEN OPEN DSKFIL$ FOR INPUT AS #3:GOTO 1000
430 OPEN DSKFIL$ FOR OUTPUT AS #3
490 DIM BUF$(128)
500 FOR J=1 TO 126
520 BUF$(J)=INPUT$(1.#1)
530 IF BUF$(J)=CHR$(26) THEN GOTO 640
540 NEXT J
```

```
550 PRINT #1.XOFF$;
560 IF LOC(1)=0 THEN K=126:GOTO 600
570 BUF$(127)=INPUT$(1.#1)
580 IF LOC(1)=0 THEN K=127:GOTO 600
585 BUF$(128)=INPUT$(1.#1)
590 K=128
600 FOR I=1 TO K
610 PRINT #3,BUF$(I):
620 NEXT I
625 PRINT #1,XON$;
630 GOTO 500
640 FOR I=1 TO J
650 PRINT #3,BUF$(I):
660 NEXT I
670 CLOSE #3:CLS:LOCATE 24,10:PRINT "* download complete *"
680 GOTO 200
800 B$=INPUT$(1.#1)
810 IF B$=XOFF$ THEN GOTO 850
820 PRINT #2.B$;
830 IF LOC(1)=0 THEN RETURN
840 GOTO 800
850 B$=INPUT$(1.#1)
860 IF B$=XON$ THEN RETURN
870 PRINT #2,B$;
880 GOTO 850
1000 WHILE NOT EOF(3)
1010 A$=INPUT$(1.#3)
1015 PRINT #1.A$;
1020 IF LOC(1)>0 THEN GOSUB 800
1030 WEND
1040 PRINT #1,CHR$(26); ctrl-z to make close file.
1050 CLOSE #3:CLS:LOCATE 23,10:PRINT "** upload complete ***;
1060 GOTO 200
9000 CLOSE #1
9010 CLOSE #2
9030 KEY ON
9040 END
```

NOTE: In the above example, for baud rates of 4800 bps and above, you must include the following line:

1014 FOR I=1 TO 10:NEXT

As mentioned earlier, when developing a communications program, you should consider both the host computer's and satellite computer's baud rates. If a "Device I/O" error occurs, this usually indicates an overrun on the hardware interface, and you should adjust your program.

Creating a System of Programs with the Runtime Module

The CHAIN with COMMON feature and the runtime module are designed for creating large systems of GW-BASIC programs that interact with each other. In this appendix, a hypothetical system will be described to show the interactions in a large system design.

The following integrated accounting system contains separate packages for general ledger, accounts payable, and accounts receivable. Entry into each package is controlled by a main menu program. The system structure is shown in Figure A-1.

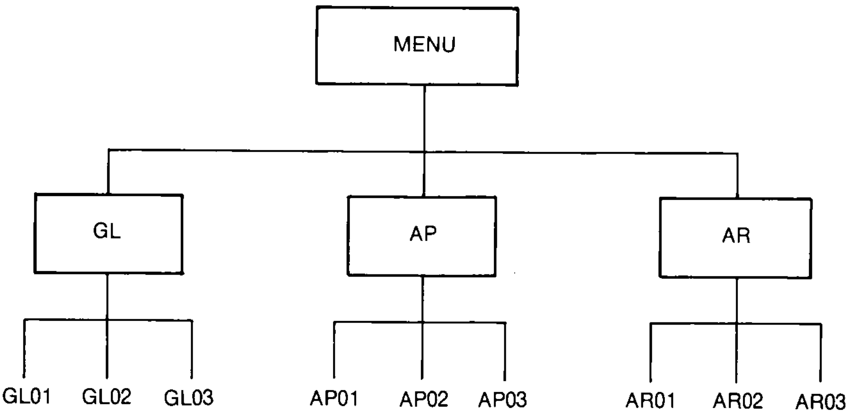


Figure A-1 Sample Program Structure

In order to use CHAIN with COMMON features effectively, it is important to logically structure the system and the COMMON information. In the system pictured above, COMMON information exists within each of the packages GL, AP, and AR. Each package contains a system of three separately compiled programs. Furthermore, there may be COMMON information between MENU and each of the packages. There may be overlapping sets of COMMON

information in the lower level programs if the different application systems always transfer to each other through a main menu.

For the preceding diagram, the use of CHAIN in each of the major programs is outlined in the following program fragments:

MENU.BAS

```
10      REM $INCLUDE: 'SYSCOM'
1000    IF MENU=1 THEN CHAIN "GL"
1010    IF MENU=2 THEN CHAIN "AP"
1020    IF MENU=3 THEN CHAIN "AR"
```

GL.BAS — General Ledger

```
10      REM $INCLUDE: 'SYSCOM'
100     REM $INCLUDE: 'GLCOMDEF'
1000    CHAIN "GL01"
1010    CHAIN "Gl02"
1020    CHAIN "GL03"
1030    IF MENU=YES THEN CHAIN "MENU"
```

AP.BAS — Accounts Payable

```
10      REM $INCLUDE: 'SYSCOM'
100     REM $INCLUDE: 'APCOMDEF'
1000    CHAIN "AP01"
1010    CHAIN "AP02"
1020    CHAIN "AP03"
1030    IF MENU=YES THEN RUN "MENU"
```

AR.BAS — Accounts Receivable

```
10      REM $INCLUDE: 'SYSCOM'
100     REM $INCLUDE: 'ARCOMDEF'
1000    CHAIN "AR01"
1010    CHAIN "AR02"
1020    CHAIN "AR03"
1030    IF MENU=YES THEN RUN "MENU"
```

Each of the lower level programs XXXY (XX=GL,AP,AR,YY=01,02,03) should CHAIN back to the package main program XX. During the execution of a CHAIN statement, the runtime module remains in memory and is not reloaded for each CHAINed program.

Unless the COMMON values are changed, modifying one program in the system requires that only that program be recompiled. For example, program GL01 can be changed and recompiled without recompiling GL02 and GL03.

Source Listing Format

The source listing file format is described and illustrated in this appendix. The discussion is keyed to the illustrated sample listing program. In this sample listing program:

Every page has a heading at the top.

The left portion of the first two lines contains the user-assigned title and subtitle, set with the metacommands $TITLE and $SUBTITLE, respectively. If these metacommands appear on the first source line, they take effect on the first page.

The right portion of the first line has the page number.

In some versions, the right side of the second line contains the date, and the right side of the third line contains the time.

The "Offset" column specifies the hexadecimal offset from the start of the .EXE file for each line of source.

The "Data" column specifies the hexadecimal offset from the start of the data segment for any data values generated by the source line.

The "Source Line" column contains a source line's line number, along with the line itself. This line number and the source file name identify runtime errors if appropriate error checking options have been used.

Two kinds of compiler messages appear in the listing: errors and warnings. A compilation with severe errors should not be linked. One with only warnings can be used to generate code, but the result may not execute correctly. Errors and warnings are listed in Appendix A of your *NCR GW-BASIC Reference Manual.* Usually, the location of the error in the source line is indicated with an up arrow (↑), followed by a two-character code. At times, however, an error in a line is not immediately detected and the error indicator may point to the end of a statement or the end of a line. This is normally the case with TC("too complex") errors.

```
GW-BASIC                                                         Page 1
Program                                                        01-01-84
                                                               12:00:00
Offset     Data      Source Line       Microsoft GW-BASIC Compiler V5-33

001A       0002      10    ' $TITLE: 'MS-BASIC' $SUBTITLE: 'Program'
001A       0002      20    DEFINT A-Z
001A       0002      30    DIM A(10,10),B(10,10),C(10,10)
001A       0002      40    PRINT "Start of program"
0034       02D8      50    ' $OCODE+
0034       02D8      60    FOR I=1 TO 10
0034       **                    L00050:
0034       **                    L00060:   MOV     AX,0001H
0037       **                              JMP     I00002
003A       02D8      70    FOR J=1 TO 10
003A       **                    I00003:
003A       **                    L00070:   MOV     AX,0001H
003D       **                              JMP     I00004
0040       02D8      80          A(I,J) = A(I-1,J-1)+B(I,J)*C(I,J)
0040       **                    I00005:
0040       **                    L00080:   MOV     AX,000BH
0043       **                              IMUL    J%
0047       **                              XCHG    AX,DI
0048       **                              ADD     DI,I%
004C       **                              SAL     DI,1
004E       **                              MOV     BX,A%-0018H[DI]
0052       **                              MOV     AX,B%[DI]
0056       **                              IMUL    C%[DI]
005A       **                              ADD     AX,BX
005C       **                              MOV     A%[DI],AX
0060       02DC      90    NEXT
0060       **                    L00090:   MOV     AX,J%
0063       **                              INC     AX
0064       **                    I00004:   MOV     J%,AX
0067       **                              CMP     WORD PTR J%,0AH
006C       **                              JNG     $-2EH
006E       02DC      100   NEXT
006E       **                    L00100:   MOV     AX,I%
0071       **                              INC     AX
0072       **                    I0002:    MOV     I%,AX
0075       **                              CMP     WORD PTR I%,0AH
007A       **                              JNG     $-42H
007C       02DC      110   ' $OCODE-
007C       02DC      120   PRINT "End of program"
0085       02DC      130   END

19216 Bytes Available
18411 Bytes Free

    0 Warning Error(s)
    0 Severe Error(s)
```

Memory Maps

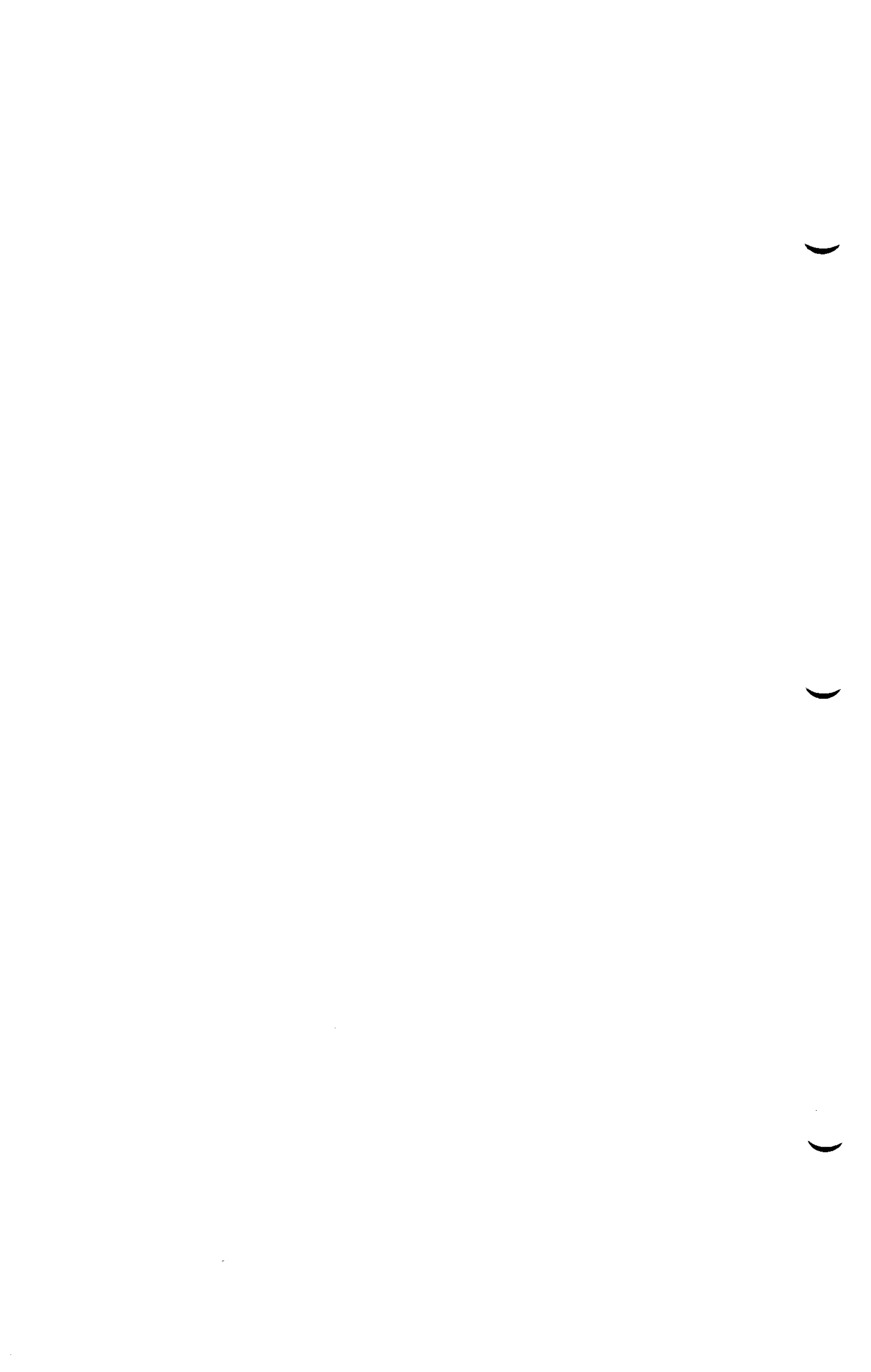
This section contains illustrations of runtime memory maps for programs linked to the two runtime libraries: BASRUNG.LIB and BASCOMG.LIB. Linking to BASRUNG.LIB causes the runtime module to be used at runtime.

RUNTIME MODULE MEMORY MAP

Memory area	Boundary
	Top of user area
space for EXE loader	
BASRUNG.EXE	Top of DS,ES,SS (64K maximum) (default is 512 bytes)
User stack	
File buffers	Top of string space
String space	(dynamic boundary)
BC_DS BC_CN BC_FT	Bottom of string space Data statements Numeric and string constants Floating-point temporaries
BC_DATA (user variables)	GW-BASIC program variables
DATA CONST (subroutines)	Optional data area for assembly language or other subroutines
COMMON (BLANK)	COMMON area (varies with program)
DATA CONST (RT_DATA)	Runtime module data and constants (fixed size is approx. 3K)
	DS,ES,SS:0000
Free memory	(if available)
CODE (assembly language or other subroutines)	Additional code (no size restriction)
BC_CODE (compiled program)	Compiled GW-BASIC code (64K maximum)
MS-DOS area Interrupt vectors	Bottom of user area
	0000:0000

ALTERNATE (BASCOMG.LIB) MEMORY MAP

Memory area	Boundary / description
	Top of memory
Space for EXE loader	Top of DS,ES,SS (64K maximum) (default is 512 bytes)
User stack	Top of string space
File buffers	(dynamic boundary)
String space	Bottom of string space EXE file loader data (64 bytes)
RUN (opt.)	Data statements Numeric and string constants Floating-point temporaries
BC_DS BC_CN BC_FT	GW-BASIC program variables
BC_DATA (user variables)	Optional data area for assembly language or other subroutines
DATA CONST (subroutines)	COMMON area (variable per program)
COMMON (BLANK)	Runtime module data and constants (fixed size is approx. 3K)
DATA CONST (RT_DATA)	DS,ES,SS:0000
Free memory	(if available)
CODE (assembly language or other subroutines)	Additional code (no size restriction)
BC_CODE (compiled program)	Compiled GW-BASIC code (64K maximum)
MS-DOS area Interrupt vectors	Bottom of user area
	0000:0000



Runtime Segment Map

The segment maps for compiled programs under NCR GW-BASIC are almost the same for versions with and without the runtime module. The table below shows both versions.

Address	**With RT Module**		**Without RT Module**	
	Segment	Class	Segment	Class
Low CS	BC_CODE	CODE	BC_CODE CODE	CODE CODE
	BC_INC BC_IDS INIT	INIT INIT INIT	BC_ICN BC_IDS INIT	INIT INIT INIT
Low DS	CONST DATA COMMON CONST DATA BC_DATA BC_FT BC_CN BC_DS RUN	RT_DATA RT_DATA BLANK CONST DATA DATA DATA DATA DATA DATA	CONST DATA COMMON CONST DATA BC_DATA BC_FT BC_CN BC_DS RUN	RT_DATA RT_DATA BLANK CONST DATA DATA DATA DATA DATA DATA
High DS	STACK	STACK	STACK	STACK
Highest memory	BASRUNG.EXE Runtime module code			

RUNTIME SEGMENT MAP

The segments BC ICN and BC IDS are block transferred to the segments BC CN and BC DS at program initialization. Just before the user program itself executes, the DS segment is moved down in

physical memory over the segments of class INIT. If the runtime module is used, then the data segment is moved to high memory under the runtime module.

All the classes and segments in the data segment (DS) are in the group DGROUP. The contents of the segments are as follows:

BC_CODE	Compiled user program
CODE	GW-BASIC runtime routines
BC_ICN	User program constants (moved to BC_CN)
BC_IDS	User program data statements (moved to BC_DS)
INIT	Disposable runtime initialization code
CONST	Runtime initialized data values
DATA	Runtime uninitialized data values
COMMON	User program COMMON area
CONST	User initialized data variables (assembly, MS-Pascal, MS-FORTRAN)
DATA	User data variables (assembly, MS-Pascal, MS-FORTRAN)
BC_DATA	User program data variables
BC_FT	User program floating-point temporaries
BC_CN	User program constants
BC_DS	User program data statements
RUN	Relocatable data segment used by the RUN statement
STACK	Stack segment required by loader (not used)

The string space and stack space are set up at initialization time. The string space uses all the available space not occupied by code and data (up to the 64K total for DS segment), except for 512 bytes reserved for the stack. In general, while an GW-BASIC Compiler program is running, the segment registers (DS, ES, and SS) are the same. CS varies depending on whether the program or runtime code is executing.

Assembly Language Routines

Note that the procedure for calling assembly language subroutines from compiled programs differs slightly from the procedure used with interpreted programs.

CALL STATEMENT

Invoking the CALL statement is the same as for the interpreter. The format is:

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

<variable name> contains the entry point of the subroutine being called. (The entry point must be declared as PUBLIC within the assembly language program that contains the subroutine.)

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

Routines written for the interpreter may be used with the compiler if they take into account the difference in string descriptors (described below) and declare the code segment as follows:

```
CODE SEGMENT BYTE PUBLIC 'CODE'
 .
 . user code here
 .
CODE ENDS
```

If an argument that is passed to the routine is a string, the argument's offset points to 4 bytes called the "string descriptor." Bytes 0 and 1 of the string descriptor contain the length of the string (0 to 32767). Bytes 2 and 3, respectively, are the lower and upper 8 bits of the string starting address in string space.

The string start address points to the first character in the string.

WARNING

Do not tamper with the compiler string descriptors, or a "String Space Corrupt" error may result.

The following demonstration program shows how an assembler routine should be written to be usuable from the compiler or the interpreter. This routine may serve as a template for user's assembler routines. If the routine is only going to be used with the compiler, then the code under interpreter switches may be removed (or vice-versa). The differences are:

1. String descriptors (see descriptions above).

2. Variables — Should be in DATA SEGMENT, GROUP DGROUP for the compiler, and CODE SEGMENT for the interpreter. The variables may be in the CODE SEGMENT for the compiler as well, but less efficient code will result because of the need for segment overrides.

title SAMPLE — Convert 16 bit integer to string

comment *

Demonstration program to show how to write assembler routines callable from BASIC in either the interpreter or the compiler implementation.

```
*

_bascom = 1    ; = 0 if this is for the interpreter

code        segment byte public 'code' ;enter exactly as shown

if          _bascom  ;if compiler, put variables in DATA
                     ;segment, otherwise it goes in
                     ;the CODE segment — MASM puts in
                     ;the segment overrides automatically

DATA        segment byte public 'rtdata' ;enter exactly as shown

endif       ;bascom
            ;any needed variables go here
flag            db  ?
separator       db     "," ;"," for US version
                           ;"." for European version
if          _bascom

DATA        ends
DGROUP GROUP    DATA ;if you have DATA this must be
present
            assume    ds:DGROUP, es:DGROUP

endif       ;bascom

            assume cs:code

publics proc      far  ;procs called from basic get far calls

;makstr —   convert 16 bit integer to string, with commas
;           properly inserted.
; usage:    ANS$ = SPC$(7)
;           'must be a temporary string, not
;           'a constant, don't do: ans$="       "
;           CALL makstr ( INT% , ANS$)
; entry:    [SP+4] - string descriptor of ANS$
;           [SP+6] - value of INT%
; exit:     number is right justified in string
;           SS,DS,ES,BP  are preserved
;           SP           is cleaned up
;           all others may be used
```

```
        PUBLIC   MAKSTR
makstr:
        push          bp
        ;Must be saved, subtracts 2 from SP
        mov           bp,sp
        ;indexing off BP uses SS, not DS
        add           bp,4+(2*2)
        ;4 byte rtn addr, 2 byte saving bp, 2 args
        ; ARG0 at [bp]
        ; ARG1 at [bp-2]
        ; ARGn at [bp-(2*n)]
        ;** get the string descriptor
        mov           si,[bp-2] ;si has string descriptor
if      __bascom      ;get string length in ax
        lodsw         ;length is word in compiler
else
        lodsb         ;length is byte in interpreter
        cbw
endif
        mov           di,ax
        lodsw         ;ax - first char in string
        dec           di
        add           di,ax ;di - last char in string

        mov           byte ptr [di],"0"
                ;just in case it's zero

        ;** get the integer data to be converted
        mov           bx,[bp]    ;bx - int%
        mov           ax,[bx]    :ax = value to be printed

        xor           cx,cx      ;initialize char counter
        mov           bx,10d     ;want base 10.

       mov flag,ah    ;is int% negative?
       or   ax,ax
       jns  ptrloop
       neg  ax

ptrloop:     ;** get digits one at a time
       xor  dx,dx     ;clear high word
       idiv bx        ;ax = quotient, dx = digit
```

```
          or   dl,dl        ;is it zero?
          jnz  printit

          or   ax,ax        ;yes, is it a leading 0?
          jz   done         ;yes, done

printit:
          call docomma ;put a comma if needed.
          add dl,"0"        ;make into ASCII char
          mov byte ptr [di],dl
          jmp short ptrloop

done:
          test byte ptr flag, 080H    ;was it negative?
          jz   reallydone
          mov byte ptr [di],"-"
               ;yes - put in a minus sign

reallydone:
          pop  bp
          ret  4            ;2 args, 2 bytes/arg

publics   endp

locals    proc near         ;for short calls

docomma:       ;do we need to put in a comma?
          inc  cl           ;cl counts chars printed
          test cl,3         ;are we on a multiple of 4?
          jnz  nocomma ;no
          mov ch,byte ptr separator
          mov [di],ch
          inc  cl
          dec  di
nocomma:ret

locals    endp

code      ends
          end
```

CALLS STATEMENT

CALLS works the same with the compiler as with the interpreter. See your *NCR GW-BASIC Reference Manual* for a description of CALLS.

USR FUNCTION

With the compiler's USR function, there is no way to pass arguments except by using POKE statements to protected memory locations that are later accessed by the assembly language subroutine. See your *NCR GW-BASIC Reference Manual* for complete information on the USR function.

INDEX

$INCLUDE metacommand, 3-1, 8-2
$LINESIZE metacommand, 8-4
$LIST metacommand, 8-3
$OPCODE metacommand, 8-3
$PAGE metacommand, 8-4
$PAGEIF metacommand, 8-4
$PAGESIZE metacommand, 8-4
$SKIP metacommand, 8-4
$SUBTITLE metacommand, 8-4
$TITLE metacommand, 8-4

/4 switch, 5-7
/A switch, 5-9
/C:<size> switch, 5-9
/D switch, 5-10
/E switch, 5-8
/N switch, 5-10
/O switch, 5-11
/R switch, 5-11
/S switch, 5-11
/T switch, 5-7
/V switch, 5-9
/W switch, 5-9
/X switch, 5-9

Arithmetic overflow check, 5-10
Array bounds check, 5-10
Assembly language subroutines, E-1

BASCOMG.LIB runtime library, 1-4
BASRUNG.LIB runtime library, 1-4

CALL statement, E-1
CALLS statement, E-6
CHAIN, A-1
COMMON, A-1
Communications, 10-1
    Control Signals, 10-3
    Get and Put functions, 10-2
    Input/Output, 10-1
    Input/Output functions, 10-2
    Sample Program, 10-4
Compiler switches, 5-5
Configuration, 5-12
Convention switches, 5-7
    /4, 5-7
    /T, 5-7

Default file specification, 5-1
Differences between interpreter and compiler, 9-1
Double precision arithmetic functions, 9-3
Double precision loop control variables, 9-3

Error handling switches, 5-8
    /E, 5-8
    /X, 5-9
Event trapping switches, 5-9
    /V, 5-9
    /W, 5-9
Exception handling switches, 5-8
    /E, 5-8
    /X, 5-9
    /V, 5-9
    /W, 5-9
Expression evaluation, 9-2

Filenaming conventions, 5-1
Floating-point calculations, 9-1

Implementation differences, 9-1
    double precision arithmetic functions, 9-3
    double precision loop control variables, 9-3
    expression evaluation, 9-2

floating-point calculations, 9-1
integer variables, 9-3
string size, 9-3
string space implementation, 9-3
Integer variables, 9-3
Invocation, 5-2

Line length, 3-2
Line number check, 5-10

Memory maps for compiler, C-1
Metacommands, 8-1
$INCLUDE, 3-1, 8-2
$LINESIZE, 8-4
$LIST, 8-3
$OCODE, 8-3
$PAGE, 8-4
$PAGEIF, 8-4
$PAGESIZE, 8-4
$SKIP, 8-4
$SUBTITLE, 8-4
$TITLE, 8-4

Operational differences, 9-1

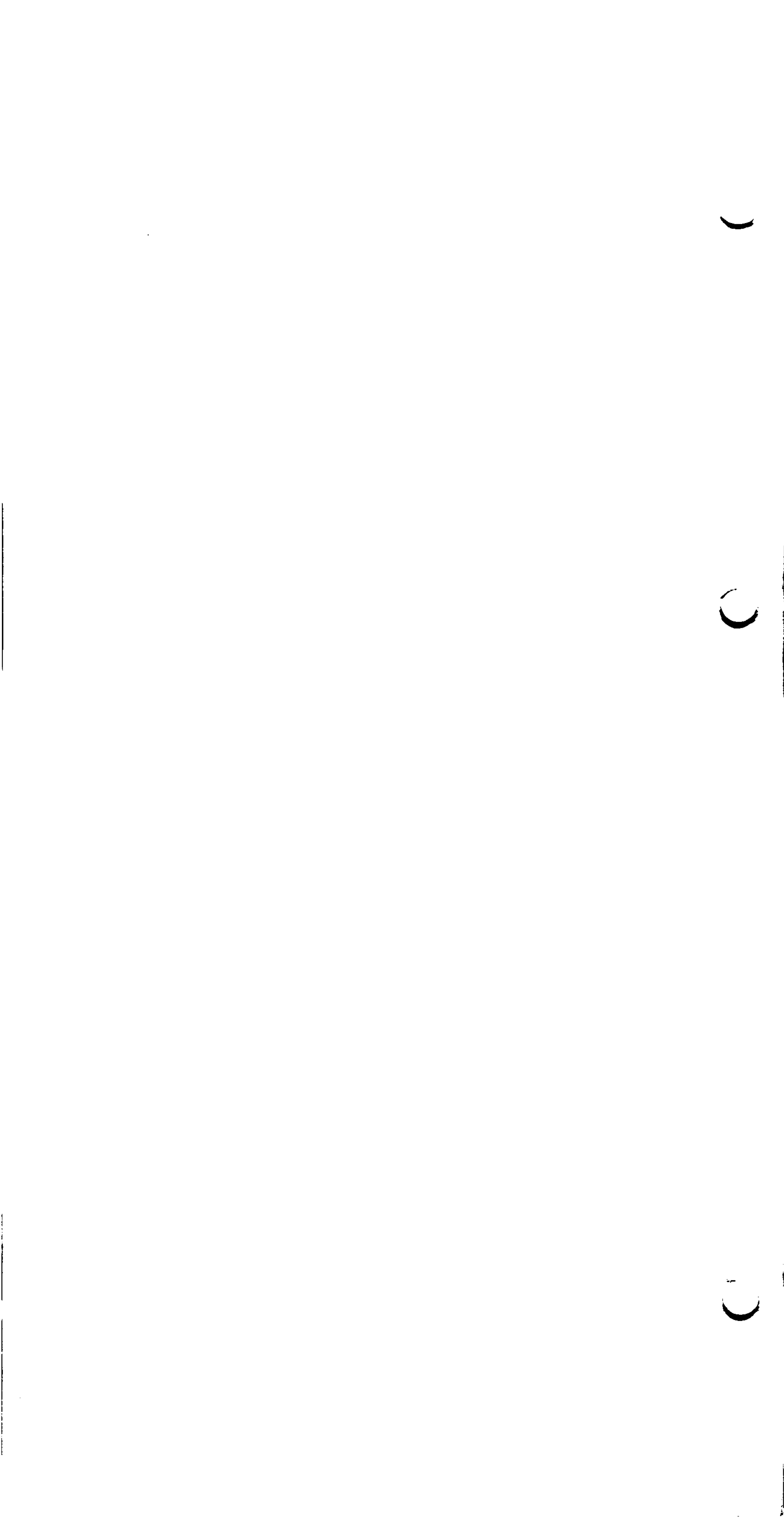
Redo message, 5-8
RESUME statement, 5-8
RETURN check, 5-10
Runtime libraries, 6-1
BASCOMG.LIB, 6-1
BASRUNG.LIB, 6-1
Runtime module, A-1
Runtime segment map, D-1

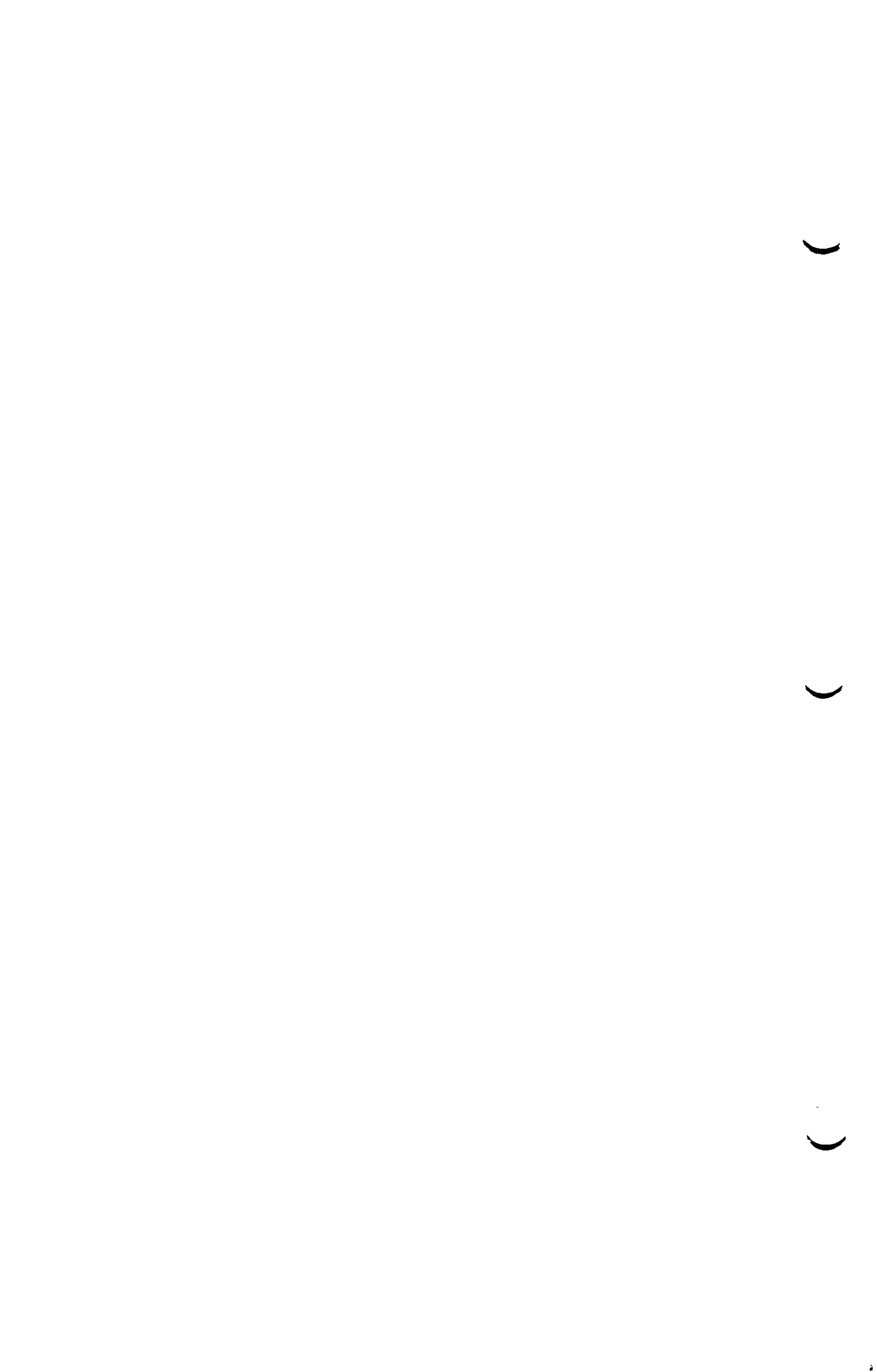
Special code switches, 5-9
/A, 5-9
/C:<size>, 5-9
/D, 5-10
/N, 5-10
/O, 5-11
/R, 5-11

/S, 5-11
String
descriptor, E-1
size, 9-3
space implementation, 9-3
Syntax notation, iv
System requirements, ii

TROFF statement, 5-10
TRON statement, 5-10

USR function, E-6





CUSTOMER PROGRAM LICENSE AGREEMENT

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THIS DISKETTE(S) PACKAGE. OPENING THIS DISKETTE(S) PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED; AND YOUR MONEY WILL BE REFUNDED.

NCR provides this Program(s) and licenses its use under these terms and conditions and under Copyright Law: You assume responsibility for the selection of the Program(s) to achieve your intended results, and for the installation, use and results obtained from the Program(s). This program is confidential, proprietary to and a trade secret of the owner, and should be safeguarded by you as such.

LICENSE

You may:

a. use the Program(s) only on a single machine at a single location;

b. copy the program into any machine readable or printed form for backup or modification purposes only, to support your use of the Program(s) on the single machine (Certain programs, however, may include mechanisms to limit or inhibit copying. They are marked "copy protected.");

c. modify the Program(s) and/or merge it into another program for your use on the single machine (Any portion of this Program(s) merged into another program will continue to be subject to the terms and conditions of this Agreement.); and

d. transfer the Program(s) and license to another party only if the other party agrees to accept the terms and conditions of this Agreement. You must advise NCR of the name and address of the other party and the other party must sign a copy of the NCR Customer Program License Agreement and have the same received by NCR. If you transfer the Program(s), you must at the same time either transfer all copies whether in printed or machine readable form to the same party or destroy any copies not transferred; this includes all modifications and portions of the Program(s) contained or merged into other programs.

You must reproduce and include any copyright notice and serial number on any copy, modification or portion merged into another program.

TERM

The license is effective until terminated. You may terminate it at any time by destroying the program together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the Program(s) together with all copies, modifications and merged portions in any form.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PROGRAM(S), OR ANY COPY, MODIFICATION OR MERGED PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION OR MERGED PORTION OF THE PROGRAM TO ANOTHER PARTY, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

EXCLUSION OF WARRANTY

THE PROGRAM(S) IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM(S) PROVE DEFECTIVE, YOU (AND NOT NCR OR ITS DEALER OR DISTRIBUTOR) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. NCR does not warrant that the functions contained in the Program(s) will meet your requirements or that the operation of the program will be uninterrupted or error free.

LIMITED WARRANTY

NCR warrants the diskette(s) on which the program is furnished to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

NCR's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette(s) not meeting NCR's "Limited Warranty" and which is returned to NCR or an authorized NCR dealer or distributor, with a copy of your receipt, or
2. if NCR or its authorized dealer or distributor is unable to deliver a replacement diskette(s) and repair is not practicable or cannot be timely made, you may terminate this Agreement by returning the program and your money will be refunded.

IN NO EVENT WILL NCR BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE DISKETTE(S) EVEN IF NCR OR AN AUTHORIZED NCR DEALER OR DISTRIBUTOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

Some states do not allow limitations on how long an implied warranty lasts, so the above exclusion may not apply to you.

Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

This warranty gives you specific legal rights and you may also have other rights which vary from state to state.

MISCELLANEOUS

You may not sublicense, assign or transfer the license or the Program(s) except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void, and will automatically terminate your license and right to use this program.

This Agreement will be governed by the laws of the State of Ohio where NCR Corporation has its principal office.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US OR BETWEEN YOU AND ANY DEALER OR DISTRIBUTOR RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Should you have any questions concerning this Agreement, you may contact NCR by writing to: NCR CORPORATION
P.O. Box 507
Dept. CSP-5
Dayton, Ohio 45409
USA

NCR CORPORATION
CUSTOMER PROGRAM LICENSE AGREEMENT ACKNOWLEDGEMENT CARD

Please complete and return this card. Keep the Customer Program License Agreement in your files.

I have read the NCR Corporation Customer Program License Agreement and agree to abide by the terms contained in it.

PRODUCT NAME : GW (TM) BASIC COMPILER
PART NUMBER : D006-0157-0000
VERSION NUMBER: 5.50 (MS-DOS)
SERIAL NUMBER : MS 557

Name ______________________________ Signature ______________________________
(Please type or print)

Company ____

Address ____

City ______________________ State ______________ Zip ______________

Country ______________________________ Date ______________________________

COMPLETE AND MAIL THE CARD BELOW

SO THAT YOU WILL BE PLACED ON OUR SOFTWARE CUSTOMER LIST.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 3 DAYTON, OHIO

POSTAGE WILL BE PAID BY ADDRESSEE

NCR CORPORATION
P.O. BOX 507
DAYTON, OHIO 45409
USA

DEPT CSP-5