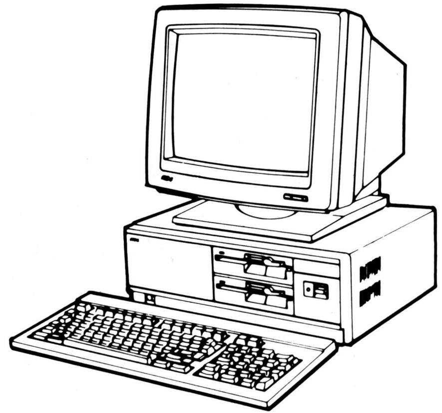
APC III



GW™-BASIC Guide for the Software Library Expander

NEC

NEC Information Systems, Inc.

819-150272-000 Rev. 00
5-85

Important Notice

(1) All rights reserved. This manual is protected by copyright. No part of this manual may be reproduced in any form whatsoever without the written permission of the coypright owner.

(2) The policy of NEC being that of continuous product improvement, contents of this manual are subject to change, from time to time, without notice.

(3) All efforts have been made to ensure that the contents of this manual are correct; however, should any errors be detected, NEC would greatly appreciate being informed.

(4) NEC can assume no responsibility for errors in this manual or their consequences.

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft® Corporation. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy GW-BASIC for the Software Library Expander (SLE) on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

SLE software is Copyright 1985© by Phoenix Software Associates Ltd.
Microsoft is a registered trademark, and MS and GW are trademarks of Microsoft Corporation.
Teletype is a registered trademark of Teletype Corporation.

Copyright 1985©

NEC Information Systems, Inc.
1414 Mass. Ave.
Boxborough, MA 01719

Printed in U.S.A.

Contents

Page

Chapter 1 Introduction

1.1 Overview 1-1
1.2 Syntax Notation 1-2
1.3 Resources for Learning BASIC 1-3

Chapter 2 Using SLE GW-BASIC

2.1 Invoking BASIC 2-1
2.2 Command Line Option Switches 2-2
2.3 Modes of Operation 2-3
2.4 Line Format 2-4
2.5 Active and Visual (Display) Pages 2-4

Chapter 3 Learning the Language

3.1 Character Set 3-1
3.1.1 Special Characters 3-1
3.1.2 Control Characters 3-3
3.2 Constants 3-4
3.2.1 String and Numeric Constants 3-4
3.2.2 Single/Double Precision Numeric Constants 3-5
3.3 Variables 3-6
3.3.1 Variable Names and Declaration Characters 3-6
3.3.2 Array Variables 3-8
3.3.3 Space Requirements 3-8
3.4 Expressions and Operators 3-9
3.4.1 Precedence of Operations 3-9
3.4.2 Arithmetic Operators 3-10
3.4.2.1 Integer Division and Modulus Arithmetic 3-12
3.4.2.2 Overflow and Division by Zero 3-13
3.4.3 Relational Operators 3-13
3.4.4 Logical Operators 3-14
3.4.5 String Operators 3-17
3.5 Type Conversion 3-18
3.6 Functions 3-19
3.6.1 Intrinsic Functions 3-19
3.6.2 User-Defined Functions 3-19

Contents

	Page
3.7 The Keyboard	3-20
3.7.1 Function Keys	3-21
3.7.2 Typewriter Keyboard	3-21
3.7.3 Numeric Keyboard	3-22

Chapter 4 Writing Programs Using the SLE GW-BASIC Editor

4.1 EDIT Command	4-1
4.2 Full Screen Editor	4-1
4.2.1 Writing Programs	4-1
4.2.2 Editing Programs	4-3
4.2.3 Logical Line Definition with INPUT	4-10
4.2.4 Editing Lines with Syntax Errors	4-10

Chapter 5 Working with Files and Devices

5.1 Default Device	5-1
5.2 Device-Independent Input/Output	5-1
5.3 Filenames and Paths	5-2
5.3.1 Filename Specifications	5-2
5.3.2 Pathnames	5-2
5.3.3 Working with Pathnames in BASIC	5-4
5.4 Re-direction of Standard Input and Standard Output	5-5
5.5 Handling Files	5-6
5.5.1 Program File Commands	5-7
5.5.2 Protecting Program Files	5-8
5.6 Data Files: Sequential and Random Access I/O	5-9
5.6.1 Sequential Files	5-9
5.6.1.1 Creating a Sequential File	5-10
5.6.1.2 Reading Data from a Sequential File	5-11
5.6.1.3 Adding Data to a Sequential File	5-12
5.6.2 Random Access Files	5-12
5.6.2.1 Creating a Random Access File	5-13
5.6.2.2 Accessing a Random Access File	5-14

Contents

	Page
5.6.2.3 Random File Operations	5-15
5.7 BASIC and Child Processes	5-19

Chapter 6 Using Advanced Features

6.1 Assembly Language Subroutines	6-1
6.1.1 Memory Allocation	6-2
6.1.2 Internal Representation	6-2
6.1.3 CALL Statement	6-3
6.1.4 CALLS Statement	6-9
6.1.5 USR Function	6-9
6.2 Event Trapping	6-10
6.2.1 ON GOSUB Statement	6-11
6.2.2 RETURN Statement	6-12

Chapter 7 Basic Commands, Functions and Statements

7.1 *ABS* Function	7-1
7.2 *ASC* Function	7-2
7.3 *ATN* Function	7-3
7.4 *AUTO* Command	7-4
7.5 *BLOAD* Command	7-5
7.6 *BSAVE* Command	7-7
7.7 *CALL* Statement	7-8
7.8 *CALLS* Statement	7-9
7.9 *CDBL* Function	7-10
7.10 *CHAIN* Statement	7-11
7.11 *CHDIR* Statement	7-15
7.12 *CHR$* Function	7-16
7.13 *CINT* Function	7-17
7.14 *CIRCLE* Statement	7-18
7.15 *CLEAR* Statement	7-20
7.16 *CLOSE* Statement	7-21
7.17 *CLS* Statement	7-23
7.18 *COLOR* Statement	7-24

Contents

Page

7.19 *COM* Statement 7-26
7.20 *COMMON* Statement 7-27
7.21 *CONT* Command 7-28
7.22 *COS* Function 7-29
7.23 *CSNG* Function 7-30
7.24 *CSRLIN* Function 7-31
7.25 *CVI, CVS, CVD* Functions 7-32
7.26 *DATA* Statement 7-33
7.27 *DATE$* Statement 7-34
7.28 *DATE$* Function 7-35
7.29 *DEF FN* Statement 7-36
7.30 *DEFINT/SNG/DBL/STR* Statements 7-38
7.31 *DEF SEG* Statement 7-39
7.32 *DEF USR* Statement 7-40
7.33 *DELETE* Command 7-41
7.34 *DIM* Statement 7-42
7.35 *DRAW* Statement 7-43
7.36 *EDIT* Command 7-46
7.37 *END* Statement 7-47
7.38 *ENVIRON* Statement 7-48
7.39 *ENVIRON$* Function 7-50
7.40 *EOF* Function 7-51
7.41 *ERASE* Statement 7-52
7.42 *ERDEV, ERDEV$* Functions 7-53
7.43 *ERR* and *ERL* Functions 7-54
7.44 *ERROR* Statement 7-55
7.45 *EXP* Function 7-57
7.46 *FIELD* Statement 7-58
7.47 *FILES* Statement 7-61
7.48 *FIX* Function 7-63
7.49 *FOR...NEXT* Statement 7-64
7.50 *FRE* Function 7-67
7.51 *GET* Statement — File I/O 7-68
7.52 *GET* Statement — Graphics 7-69
7.53 *GOSUB...RETURN* Statements 7-71

Contents

Page

7.54 *GOTO* Statement 7-73
7.55 *HEX$* Function 7-74
7.56 *IF . . . THEN [. . . ELSE]/IF . . . GOTO* Statements....... 7-75
7.57 *INKEYS$* Function 7-78
7.58 *INP* Function 7-79
7.59 *INPUT* Statement 7-80
7.60 *INPUT#* Statement 7-82
7.61 *INPUT$* Function 7-83
7.62 *INSTR* Function 7-85
7.63 *INT* Function 7-86
7.64 *IOCTL* Statement 7-87
7.65 *IOCTL$* Function 7-88
7.66 *KEY* Statement 7-89
7.67 *KEY(n)* Statement 7-91
7.68 *KILL* Statement 7-93
7.69 *LEFT$* Function 7-95
7.70 *LEN* Function 7-96
7.71 *LET* Statement 7-97
7.72 *LINE* Statement 7-98
7.73 *LINE INPUT* Statement 7-101
7.74 *LINE INPUT#* Statement 7-102
7.75 *LIST* Command 7-104
7.76 *LLIST* Command 7-106
7.77 *LOAD* Command 7-107
7.78 *LOC* Function 7-108
7.79 *LOCATE* Statement 7-109
7.80 *LOF* Function 7-111
7.81 *LOG* Function 7-112
7.82 *LPOS* Function 7-113
7.83 *LPRINT* and *LPRINT USING* Statements 7-114
7.84 *LSET* and *RSET* Statements 7-115
7.85 *MERGE* Command 7-116
7.86 *MID$* Statement 7-117
7.87 *MID$* Function 7-118
7.88 *MKDIR* Statement 7-119

Contents

Page

7.89 *MKI$, MKS$, MKD$* Functions 7-120
7.90 *NAME* Statement 7-121
7.91 *NEW* Command 7-122
7.92 *OCT$* Function 7-123
7.93 *ON COM* Statement 7-124
7.94 *ON ERROR GOTO* Statement 7-126
7.95 *ON...GOSUB* and *ON...GOTO* Statements 7-127
7.96 *ON KEY* Statement 7-128
7.97 *ON TIMER* Statement 7-132
7.98 *OPEN* Statement 7-134
7.99 *OPEN COM* Statement 7-138
7.100 *OPTION BASE* Statement 7-141
7.101 *OUT* Statement 7-142
7.102 *PAINT* Statement 7-143
7.103 *PEEK* Function 7-147
7.104 *PMAP* Function 7-148
7.105 *POINT* Function 7-150
7.106 *POKE* Statement 7-152
7.107 *POS* Function 7-153
7.108 *PRESET* Statement 7-154
7.109 *PRINT* Statement 7-155
7.110 *PRINT USING* Statement 7-158
7.111 *PRINT#* and *PRINT# USING* Statements 7-163
7.112 *PSET* Statement 7-166
7.113 *PUT* Statement — File I/O 7-168
7.114 *PUT* Statement — Graphics 7-169
7.115 *RANDOMIZE* Statement 7-172
7.116 *READ* Statement 7-174
7.117 *REM* Statement 7-176
7.118 *RENUM* Command 7-178
7.119 *RESET* Command 7-180
7.120 *RESTORE* Statement 7-181
7.121 *RESUME* Statement 7-182
7.122 *RETURN* Statement 7-184
7.123 *RIGHT$* Function 7-185

Contents

Page

7.124 *RMDIR* Statement 7-186
7.125 *RND* Function 7-187
7.126 *RUN* Statement/Command 7-188
7.127 *SAVE* Command 7-189
7.128 *SCREEN* Statement 7-190
7.129 *SCREEN* Function 7-192
7.130 *SGN* Function 7-194
7.131 *SHELL* Statement 7-195
7.132 *SIN* Function 7-197
7.133 *SPACE$* Function 7-198
7.134 *SPC* Function 7-199
7.135 *SQR* Function 7-200
7.136 *STOP* Statement 7-201
7.137 *STR$* Function 7-202
7.138 *STRING$* Function 7-203
7.139 *SWAP* Statement 7-204
7.140 *SYSTEM* Command 7-205
7.141 *TAB* Function 7-206
7.142 *TAN* Function 7-207
7.143 *TIME$* Statement 7-208
7.144 *TIME$* Function 7-209
7.145 *TIMER* Function 7-210
7-146 *TIMER ON, TIMER OFF, TIMER STOP* Statements ... 7-211
7.147 *TRON/TROFF* Statements/Commands 7-212
7.148 *USR* Function 7-213
7.149 *VAL* Function 7-216
7.150 *VARPTR* Function 7-217
7.151 *VARPTR$* Function 7-219
7.152 *VIEW* Statement 7-221
7.153 *WAIT* Statement 7-224
7.154 *WHILE . . . WEND* Statements 7-225
7.155 *WIDTH* Statement 7-227
7.156 *WINDOW* Statement 7-229
7.157 *WRITE* Statement 7-230
7.158 *WRITE#* Statement 7-231

Contents

Appendix A ASCII Character Codes

Appendix B Error Codes and Error Messages

Appendix C SLE GW-BASIC Reserved Words

Appendix D Mathematical Functions Not Intrinsic to SLE GW-BASIC

Appendix E Keyboard Diagram and Scan Codes

Figures

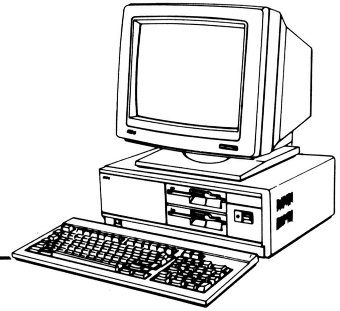
Figure	Title	Page
3-1	APC III SLE Keyboard	3-20
3-2	SLE GW-BASIC Function Key Template	3-21
3-3	Typewriter Keyboard	3-21
3-4	SLE GW-BASIC Keypad Template	3-22
6-1	Stack Layout When CALL Statement is Activated	6-4
6-2	Stack Layout During Execution of a CALL Statement	6-5

Tables

Table	Title	Page
3-1	SLE GW-Basic Relational Operators Truth Table	3-14
4-1	Special Program Editor Keys	4-3

Chapter 1

Introduction



In 1975, Microsoft wrote the first BASIC interpreter for microcomputers. Today, Microsoft BASIC has well over 1,000,000 installations and is used in many operating environments. It's the BASIC you will find on all of the most popular microcomputers. Many users, manufacturers, and software vendors have written application programs in Microsoft® BASIC.

The BASIC interpreter is a general-purpose programming language: it is effective for many applications, including business, science, games, and education. It is interactive; that is, without writing a program, a user can perform processes, calculations, and program testing.

The Software Library Expander (SLE) GW™-BASIC meets the requirements for the ANSI subset standard for BASIC. In addition, SLE GW-BASIC Interpreter has sophisticated screen handling, graphics, and structured programming features that are especially suited for application development.

1.1 OVERVIEW

SLE GW-BASIC includes several features not found in other BASICs, and has been designed to take advantage of the Software Library Expander (SLE) MS™-DOS environment to enhance programming power.

Some of the new features and improvements over GW-BASIC 1.0 are:

- Re-direction of Standard Input (INPUT, LINE INPUT) and Standard Output (PRINT)
- Character Device support which allows BASIC to initialize and communicate with user-installed devices
- Improved Disk I/O facilities for handling larger files
- SHELL which allows COMMAND or Child processes to run

- Multi-level directories for better disk organization
- Directory management (MKDIR/CHDIR/RMDIR)
- Improved Graphics: Line Clipping, VIEW, WINDOW
- Screen Editor enhancements including text window support
- Additional Event Trapping: TIMER
- User definable Keyboard trapping
- More precise error reporting with the new system functions: ERDEV and ERDEV$
- Double Precision Transcendentals (optional with the /D switch)
- More precise control of BASIC's memory allocation for user routines with the /M: switch

1.2 SYNTAX NOTATION

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

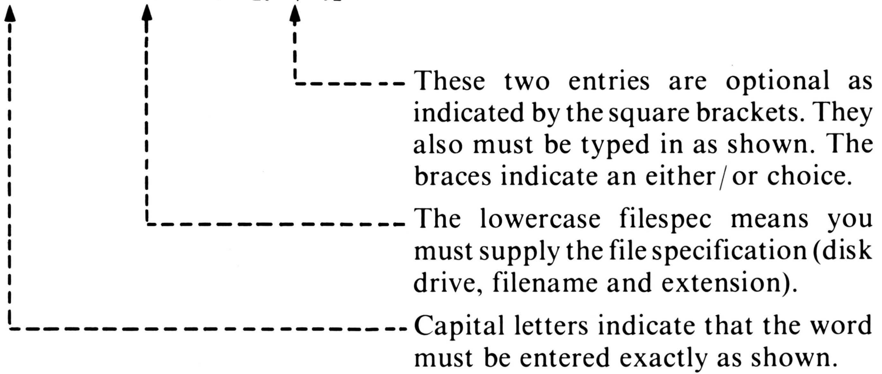
[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user-entered data. When the angle brackets enclose lowercase text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose uppercase text, the user must press 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.

Examples

Command Line Explanation

SAVE <filespec>, [{A|P}]



1.3 RESOURCES FOR LEARNING BASIC

This manual provides complete instructions for using Microsoft BASIC. However, no training material for BASIC programming has been provided. If you are new to BASIC or need help in learning programming, we suggest you read one of the following:

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

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

Billings, Karen and Moursund, David. *Are You Computer Literate?* Beaverton, Oregon: Dilithium Press, 1979.

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

Microsoft is a registered trademark and MS and GW are trademarks of Microsoft Corporation.

/S:<lrecl>

This switch is ignored unless the /I switch is specified on the command line. Please refer to the /I switch documentation below.

If this switch and the /I switch are present, then the maximum record size allowed for use with random files is set to <lrecl>. NOTE: the record size option to the OPEN statement cannot exceed this value. If the /S: option is omitted, the record size defaults to 128 bytes.

/C:<buffer size>

If present, this switch controls RS232 Communications. If RS232 cards are present, /C:0 disables RS232 support. Any subsequent I/O attempts will result in a "Device Unavailable" error. Specifying /C:<n> allocates space for communications buffers. The amount of space allocated is dependent on the machine-specific portion of SLE GW-BASIC.

/D

If present, this switch causes the Double Precision Transcendental math package to remain resident. If omitted, this package is discarded and the space is freed for program use.

/I

SLE GW-BASIC is able to dynamically allocate space required to support file operations. For this reason, SLE GW-GASIC does not need to support the /S and /F switches. However, certain applications have been written in such a manner that certain BASIC internal data structures must be static. In order to provide compatibility with these BASIC programs, SLE GW-BASIC will statically allocate space required for file operations based on the /S and /F switches when the /I switch is specified.

/M:[<highest memory location>] [,<max block size>]

When present, this switch sets the highest memory location that will be used by BASIC. BASIC will attempt to allocate 64K of memory for the data and stack segment. If machine language subroutines are to be used with BASIC programs, use the /M: switch to set the highest location that BASIC can use. When omitted or 0, BASIC attempts to allocate all it can up to a maximum of 65536 bytes.

If you intend to load things above the highest location that BASIC can use, then use the optional parameter <maximum block size> to preserve space for them. This is necessary if you intend to use the SHELL statement (see Section 7.131). Failure to do so will result in COMMAND being loaded on top of your routines when a SHELL statement is executed.

<maximum block size> must be in paragraphs (byte multiples of 16). When omitted, &H1000 (4096) is assumed. This allocates 65536 bytes (65536= 4096 × 16) for BASIC's data and stack segment. For example, if you wanted 65536 bytes for BASIC and 256 bytes for machine language subroutines, then use /M:,&H1010 (4096 paragraphs for BASIC + 16 paragraphs for your routines).

This option can also be used to shrink the BASIC block in order to free more memory for shelling other programs. /M:,2048 allocates 32768 bytes for data and stack. /M:32000,2048 allocates 32768 bytes maximum, but BASIC will only use the lower 32000. This leaves 768 bytes for the user.

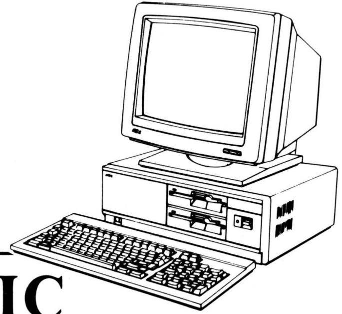
NOTE

<number of files>, <lrecl>, <buffer size>, <highest memory location>, and <maximum block size> are numbers that may be decimal, octal (preceded by &O), or hexadecimal (preceded by &H).

Example:

A>BASICA PAYROLL	Use 64K of memory and 3 files, load and execute PAYROLL.BAS.
A>BASICA INVENT/I/F:6	Use 64K of memory and 6 files, load and execute INVENT.BAS.
A>BASICA /C:0/M:32768	Disable RS232 support and use only the first 32K of memory. The memory above that is free for the user.

A>BASICA /I/F:4/S:512	Use 4 files and allow a maximum record length of 512 bytes.
A>BASICA TTY/C:512	Use 64K of memory and 3 files. Allocate 512 bytes to RS232 receive buffers and 128 bytes to transmit buffers, load and execute TTY.BAS.



Chapter 2

Using SLE GW-BASIC

2.1 INVOKING BASIC

To begin operating the Software Library Expander (SLE) GW-BASIC Interpreter, load the Software Library Expander (SLE) MS-DOS operating system and then enter:

```
BASICA
```

To begin operating a specific program as soon as BASIC has started, load the operating system and enter:

```
BASICA <filespec>
```

where <filespec> is a filename preceded by an optional device designator, and followed by an optional extension name.

For example, to start the program FILE.BAS which is on disk drive A:, enter:

```
BASICA A:FILE.[BAS]
```

2.2 COMMAND LINE OPTION SWITCHES

The BASIC operating environment may be altered somewhat by specifying option switches following BASIC on the command line. The format of BASIC's command line is:

```
BASICA [<stdin]
       [>stdout]
       [<filespec>]
       [/C:<buffer size>]
       [/D]
       [/F:<number of files>]
       [/I]
       [/M:[<highest memory location>]
           [,<maximum block size>]]
       [/S:<lrecl>]
```

WHERE:

<stdin

BASIC input is redirected from the file specified by stdin. When present, this syntax must appear before any switches. Note that the less-than character "<" is literally that character, and not an angle bracket indicating a required argument.

>stdout

BASIC output is directed to the screen and to the file specified by stdout. When present, this syntax must appear before any switches. If two greater-than signs appear (">>"), the output is appended to an existing output file. If an existing file is to be written to, this is the way to prevent that file from being overwritten. Note that the greater-than character ">" is literally that character, and not an angle bracket indicating a required argument.

<filespec>

This is the file specification of a BASIC program. If <filespec> is present, BASIC proceeds as if a RUN <filespec> command were given after initialization is complete. This allows BASIC programs to be initiated by a batch file by putting this form of the command line in an AUTOEXEC.BAT file. Programs run in this manner will need to exit via the SYSTEM statement in order to allow the next command from the AUTOEXEC.BAT file to be executed.

/F:<number of files>

This switch is ignored unless the /I switch is specified on the command line. Please refer to the /I switch documentation below.

If this switch and the /I switch are present, the maximum number of files that may be open simultaneously during the execution of a BASIC program is set to <number of files>. Each file requires 62 bytes for the File Control Block (FCB) plus 128 bytes for the data buffer. The data buffer size may be altered via the /S: option switch. If the /F: option is omitted, the number of files is set to 3.

The number of open files that SLE MS-DOS supports depends upon the value of the FILES= parameter in the CONFIG.SYS file. It is recommended that FILES=10 for BASIC. Keep in mind that the first 3 are taken by Stdin, Stdout, Stderr, Stdaux, and Stdprn. One additional handle is needed by BASIC for LOAD, SAVE, CHAIN, NAME, and MERGE. This leaves 6 for BASIC File I/O, thus /F:6 is the maximum supported by SLE MS-DOS when FILES=10 appears in the CONFIG.SYS file.

Attempting to OPEN a file after all the file handles have been exhausted will result in a "Too many files" error.

2.3 MODES OF OPERATION

The SLE GW-BASIC Interpreter may be used in either of two modes: direct mode or indirect mode.

In direct mode, statements and commands are executed as they are entered. They are not preceded by line numbers. After each direct statement followed by a carriage return, the screen will display the "Ok" prompt. 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 the SLE 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 later be stored in memory. The program stored in memory is executed by entering the RUN command.

2.4 LINE FORMAT

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

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

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

An SLE 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.

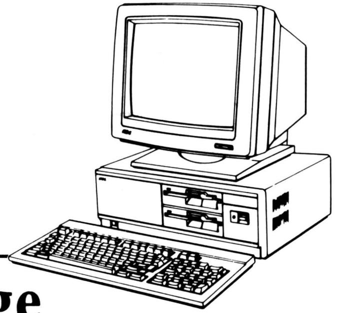
A line may contain a maximum of 255 characters.

With the interpreter, you can extend a logical line over more than one physical line by entering a <linefeed>. <linefeed> lets you continue typing a logical line on the next physical line without entering a <carriage return>. Alternatively, you may type up to 255 characters on a logical line without issuing either a line feed or a carriage return; the text is wrapped and continues on the next physical line.

A period (.) may be used in EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

2.5 ACTIVE AND VISUAL (DISPLAY) PAGES

The size of these pages is set by the SCREEN statement. (See SCREEN Statement, Section 7.128.)



Chapter 3
Learning the Language

Like any language, BASIC has an alphabet and common phrases. This chapter presents the BASIC character set, and the rules for the constants, variables, and expressions that the programming language uses.

3.1 CHARACTER SET

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

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

The SLE GW-BASIC numeric characters are the digits 0 through 9. The alphabetic characters A, B, C, D, E, and F may be used as part of hexadecimal numbers.

3.1.1 Special Characters

The following special characters and terminal keys are recognized by SLE 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

CHARACTER	ACTION
%	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
<escape>	Erases entire logical line from screen, but not from program memory.
<tab>	Moves print position to the next tab stop. Tab stops are set every eight columns. When in insert mode eight spaces are inserted. When in overtype mode the cursor is just moved eight character spaces.
<return>	Terminates input of a line.

3.1.2 Control Characters

SLE GW-BASIC supports the following control characters:

CONTROL CHARACTER	ACTION
Control-B or Control-<left arrow>	Moves cursor to previous word.
Control-Break/Stop	With the interpreter, interrupts program execution and returns to BASIC command level.
Control-E or Control-End	Clears to end of line.
Control-F or Control-<right arrow>	Moves cursor to the next word.
Control-H or Backspace	Deletes the last character typed.
Control-I or Tab	Tabs to the next tab stop. Tab stops are set every eight columns.
Control-J	Extends a logical line over more than one physical line.
Control-K or Home	Sends cursor to home location.
Control-L or Control-Clear/Home	Clears the screen.
Control-N or End	Moves cursor to the end of the line.

CONTROL CHARACTER	ACTION
Control-R or Ins	Toggles the insert and typeover modes.
Control-Num Lock	Suspends program execution.
Control-T	Updates the function key display line.

3.2 CONSTANTS

Constants are the values that cannot be changed during execution. There are two types of constants: string and numeric.

3.2.1 String and Numeric Constants

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

Examples:

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

Numeric constants are positive or negative numbers. SLE 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.)

4. Hex constants

   Hexadecimal numbers, denoted by the prefix &H. Hex constants may be no greater than decimal 65535.

   Examples

   &H76
   &H32F

5. Octal constants

   Octal numbers, denoted by the prefix &0 or &. Octal constants may not exceed decimal 65535.

   Examples:

   &0347
   &1234

3.2.2 Single/Double Precision 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

3.3 VARIABLES

Variables are names used to represent values used in an SLE GW-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).

3.3.1 Variable Names and Declaration Characters

SLE 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, with one exception: no variable may start with the letters USR. For example, the variable USRNAM$ will generate a syntax error. Reserved words include all SLE GW-BASIC commands, statements, function names, and operator names. If a variable begins with FN, it is assumed to be a call to a user-defined function.

Variables may represent either a numeric value or a string. String variable names are 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 be declared as 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 variable name is single precision. However, if a number specified in a program has too many significant digits to be represented by a single precision number, it will be represented as a double precision number, and the " " which signifies double precision will follow the number in the program listing.

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.

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

Examples of SLE 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.

The default variable type may be selectively changed by using the SLE GW-BASIC statements DEFINT, DEFSTR, DEFDBL, and DEFSNG. These statements are described in detail in Section 7.30.

3.3.2 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 has nameas 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. The maximum amount of space that may be taken for an array is 64K.

3.3.3 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	Number of bytes
	Integer	2
	Single precision	4
	Double precision	8
	String	3

Arrays	Type	Number of bytes per element
	Integer	2 per element
	Single precision	4 per element
	Double precision	8 per element
	String	3 per element

3.4 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. SLE GW-BASIC operators may be divided into three categories:

1. Arithmetic
2. Relational
3. Logical

Each category is described in the following sections.

3.4.1 Precedence of Operations

The SLE GW-BASIC operators have an order of precedence; that is, when several operations take place within the same program statement, certain kinds of operations will be performed before others. If the operations are of the same level of precedence, the first to be executed will be the leftmost, and the last, the rightmost. The following is the order in which operations are executed.

1. Exponentiation
2. Negation
3. Multiplication & Division
4. Integer Division
5. Modulus Arithmetic
6. Addition & Subtraction
7. Relational Operators
8. NOT
9. AND
10. OR & XOR
11. EQV
12. IMP

3.4.2 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 MOD 4=2 (10/4=2 with remainder 2)
+, -	Addition, Subtraction	X+Y

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.

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

ALGEBRAIC EXPRESSION	BASIC EXPRESSION
X+2Y	X+Y*2
$\frac{X-Y}{Z}$	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.

3.4.2.1 INTEGER DIVISION AND MODULUS ARITHMETIC

In addition to the six standard operators (addition, subtraction, multiplication, division, negation, and exponentiation), SLE 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:

```
100 LET DIV1 = 10\4
200 LET DIV2 = 25.68\6.99
300 PRINT DIV1, DIV2
```

will yield

```
2        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 PRINT 10.4 MOD 4, 25.68 MOD 6.99
```

will yield

```
2        5
```

because (10/4=2 with a remainder 2) and (26/7=3 with a remainder 5).

3.4.2.2 OVERFLOW AND DIVISION BY ZERO

If division by zero is encountered during the evaluation of an expression, a "Division by zero" error message is displayed. Machine infinity (the largest number that 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.

3.4.3 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 Statement, Section 7.56.)

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 7.71.)

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
```

3.4.4 Logical Operators

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.

Table 3-1 SLE GW-BASIC Relational Operators Truth Table

OPERATION	VALUE	VALUE	RESULT
NOT			
	X		NOT X
	T		F
	F		T
AND			
	X	Y	X AND Y
	T	T	T
	T	F	F
	F	T	F
	F	F	F
OR			
	X	Y	X OR Y
	T	T	T
	T	F	T
	F	T	T
	F	F	F

Table 3-1 SLE GW-BASIC Relational Operators Truth Table (cont'd)

OPERATION	VALUE	VALUE	RESULT
XOR			
	X	Y	X XOR Y
	T	T	F
	T	F	T
	F	T	T
	F	F	F
EQV			
	X	Y	X EQV Y
	T	T	T
	T	F	F
	F	T	F
	F	F	T
IMP			
	X	Y	X IMP Y
	T	T	T
	T	F	F
	F	T	T
	F	F	T

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF Statements, Section 7.56).

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, all using decimal numbers, demonstrate how the logical operators work.

Example	Explanation
63 AND 16 = 16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16.
15 AND 14 = 14	15 = binary 1111 and 14 = binary 1110, so 15 and 14 = 14 (binary 1110).
-1 AND 8 = 8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8.
4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110).
10 OR 10 = 10	10 = binary 1010, so 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.

3.4.5 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 the trailing blanks are significant.

For example:

```
“AA”              is less than       “AB”
“FILENAME”        is equal to        “FILENAME”
“X&”              is greater than    “X#”
  (because # comes before &)
“CL   ”           is greater than    “CL”
  (because of the trailing space)
“kg”              is greater than    “KG”
“SMYTH”           is less than       “SMYTHE”
B$                is less than       “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.

3.5 TYPE CONVERSION

When necessary, SLE GW-BASIC will convert a numeric constant from one type to another. The following rules and examples apply to conversions.

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

   Example:

   ```
   10 PERCENT%=23.42
   20 PRINT PERCENT%
   ```

   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 as that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

   Examples:

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

   will yield

   ```
   .8571428571428571
   ```

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

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

   will yield

   ```
   .857143
   ```

   The arithmetic was performed in double precision, and the result is rounded to single precision and returned to DEDUCTION (single precision variable), and printed.

3. Logical operators (see Section 3.4.4) 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 CASH%=55.88
   20 PRINT CASH%
   ```

   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
   ```

3.6 FUNCTIONS

SLE GW-BASIC incorporates two kinds of functions: intrinsic and user-defined.

3.6.1 Intrinsic Functions

When a function is used in an expression, it calls a predetermined operation that is to be performed on an operand. SLE GW-BASIC has functional operators that reside in the system, such as SQR (square root) or SIN (sine), and these resident functions are called "intrinsic functions".

3.6.2 User-Defined Functions

SLE GW-BASIC also allows "user-defined" functions that are written by the programmer. See DEF FN Statement, Section 7.29.

3.7 THE KEYBOARD

The keyboard is divided into three general areas:

- Twelve function keys, labeled PF1 through PF12, are on the up side of the keyboard.
- The "typewriter" area is in the middle. This is where you find the regular letter and number keys.
- The numeric keypad, similar to a calculator keyboard, is on the right side.

All the keys, in all three areas of the keyboard, are typematic. That means they repeat as long as you hold them down. Figure 3-1 contains the APC III SLE keyboard layout including the function key and keypad templates. The following sections explain each of the keyboard areas in more detail.

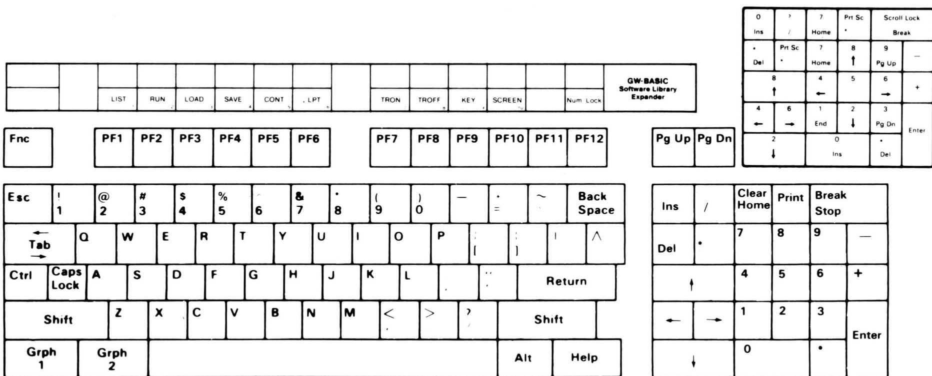


Figure 3-1 APC III SLE Keyboard

3.7.1 Function Keys

The function keys 1-10 can be used:

- As "soft keys." That is, you can set each key to automatically type any sequence of characters. In fact, some frequently used commands have already been assigned to these keys. You may change these if you wish. Refer to "KEY Statement".
- As program interrupts through use of the ON KEY statement. See "ON KEY(n) Statement".

The function key PF11 is not used by SLE GW-BASIC. Function key PF12 serves as a numeric lock switch (Num Lock) for the numeric keypad. Each time you start a new SLE GW-BASIC session, Num Lock is off. By pressing Num Lock, you can lock and unlock the numeric keypad.

Figure 3-2 is the function key template used with SLE GW-BASIC.

Figure 3-2 SLE GW-BASIC Function Key Template

3.7.2 Typewriter Keyboard

The typewriter area of the keyboard behaves much like a standard typewriter. All the letters are there, in their usual places. The numbers 0 through 9 are on the top row, along with some special characters.

Figure 3-3 Typewriter Keyboard

3.7.3 Numeric Keypad

The numeric keypad has two modes of operation; Num Lock Off and Num Lock On. Pressing Num Lock (PF12) switches between the two modes. Figure 3-3 is the keypad template used with SLE GW-BASIC. The top part of each key represents the key's function when Num Lock is on, and the bottom part represents the key's function when Num Lock is off. There are seven keys that are the same in either mode; Prt Sc/* (2), ?//, Scroll Lock/Break, -, +, and Enter. The first four keys operate the same as typewriter keys. To get the function on the bottom of the key, simply press the key. To get the function on the top of the key, press Shift and the key.

<table>
<tr><td>0
Ins</td><td>?
/</td><td>7
Home</td><td>Prt Sc
*</td><td colspan="2">Scroll Lock
Break</td></tr>
<tr><td>•
Del</td><td>Prt Sc
*</td><td>7
Home</td><td>8
↑</td><td>9
Pg Up</td><td>—</td></tr>
<tr><td colspan="2">8
↑</td><td>4
←</td><td>5</td><td>6
→</td><td>+</td></tr>
<tr><td>4
←</td><td>6
→</td><td>1
End</td><td>2
↓</td><td>3
Pg Dn</td><td rowspan="2">Enter</td></tr>
<tr><td colspan="2">2
↓</td><td colspan="2">0
Ins</td><td>•
Del</td></tr>
</table>

Figure 3-4 SLE GW-BASIC Keypad Template

Chapter 4

Writing Programs Using the SLE GW-BASIC Editor

SLE GW-BASIC provides two ways to enter and edit text: you can issue an EDIT command to place you in edit mode or use the full screen editor.

4.1 EDIT COMMAND

The EDIT command places the cursor on a specified line so that changes can be made to the line. See EDIT command, Section 7.36.

4.2 FULL SCREEN EDITOR

The full screen editor gives you immediate visual feedback, so that program text is entered in a "what you see is what you get" manner. If the user has a program listing on the screen, the cursor can be moved to a program line, the line edited, and the change entered by pressing the return key. This time-saving capability is made possible by special keys for cursor movement, character insertion and deletion, and line or screen erasure. Specific functions and key assignments are discussed in the following sections.

With the full screen editor, you can move quickly around the screen, making corrections where necessary. The changes are entered by placing the cursor on the changed line and pressing <RETURN>.

When input processes are directed from the screen, the user may use the full-screen editor features in responding to INPUT and LINE INPUT statements.

4.2.1 Writing Programs

You are using the full screen editor any time between the interpreter's "OK" prompt and the execution of a RUN command. Any line of text that is entered is processed by the editor. Any line of text that begins with a number is considered a program statement.

It is possible to extend a logical line over more than one physical line by continuing typing beyond the last column of the screen, or by pressing Control-J. The editor wraps the logical line so that it continues on the next physical line. A carriage return signals the end of the logical line; when a carriage return is entered, the entire logical line is passed to SLE GW-BASIC. Up to 255 characters may be present in one logical line.

Program statements are processed by the editor in one of the following ways:

1. A new line is added to the program. This occurs if the line number is valid (0 through 65529) and at least one non-blank character follows the line number.

2. An existing line is modified. This occurs if the line number matches that of an existing line in the program. The existing line is replaced with the text of the new line.

3. An existing line is deleted. This occurs if the line contains only the line number, and the number matches that of an existing line.

4. The statements are passed to the command scanner for interpretation (i.e., the statement is executed).

5. An error is produced.

   If an attempt is made to delete a non-existent line, an "Undefined line" error message is displayed.

   If program memory is exhausted, and a line is added to the program, an "Out of memory" error message is displayed, and the line is not added.

More than one statement may be placed on a line. If this is done, the statements must be separated by a colon (:). The colon need not be surrounded by spaces.

4.2.2 Editing Programs

Use the LIST command to display an entire program or range of lines on the screen so that they can be edited with the full screen editor. Text can then be modified by moving the cursor to the place where the change is needed and then performing one of the following actions:

1. Typing over existing characters
2. Deleting characters to the right of the cursor
3. Deleting characters to the left of the cursor
4. Inserting characters
5. Appending characters to the end of the logical line

These actions are performed by special keys assigned to the various full screen editor functions.

Changes to a line are recorded when a carriage return is entered while the cursor is somewhere on that line. The carriage return enters all changes for that logical line, and up to the 255 character line limitation, no matter how many physical lines are included and no matter where the cursor is located on the line. Table 4-1 describes the editing program keys.

Table 4-1 Special Program Editor Keys

KEY(S)	FUNCTION
Shift	Capital letters and the special characters are displayed by holding down either of the Shift keys and pressing the desired key.
Caps Lock	This keyboard does not have a normal Shift Lock key. The Caps Lock key is similar to a Shift Lock key, but it only gives you capital letters, and will not give you the uppershift characters on the numeric or other keys. After you press this key, you will continue to get capital letters until you press it again. You can get lowercase letters when in Caps Lock state by pressing and holding one of the Shift keys. When you release the Shift key, you'll go back to Caps Lock state.

Table 4-1 Special Program Editor Keys (cont'd)

KEY(S)	FUNCTION
Ctrl	The Ctrl key is also used to enter certain codes and characters not otherwise available from the keyboard. You also use the Ctrl key together with other keys when you edit programs with the program editor.
Alt	The Alt key enables easy entry of BASIC statement keywords. This key allows you to type an entire BASIC keyword with a single keystroke. The BASIC keyword is typed when the Alt key is held down while one of the alphabetic keys A-Z is pressed. Keywords associated with each letter are summarized below. Letters not having reserved words are noted by "(no word)." A AUTO B BSAVE C COLOR D DELETE E ELSE F FOR G GOTO H HEX$ I INPUT J (no word) K KEY L LOCATE M MID$ N NEXT O OPEN P PRINT Q (no word) R RUN S SCREEN T THEN U USING V VAL W WIDTH X XOR Y (no word) Z (no word) The Alt key is also used with the keys on the numeric keypad to enter characters not found on the keys. This is done by holding down the Alt key and typing the three-digit ASCII code for the character.

Table 4-1 Special Program Editor Keys (cont'd)

KEY(S)	FUNCTION
Ctrl-Break/Stop	Ctrl-Break/Stop interrupts program execution at the next BASIC instruction and returns to BASIC command level. It is also used to exit AUTO line numbering mode.
Ctrl-Numlock (PF12)	Ctrl-S sends the computer into a *pause* state. This can be used to temporarily halt printing or program listing. The pause continues until Ctrl-S is pressed again.
Ctrl-E or Ctrl-End	Erases to the end of logical line from the current cursor position. All physical screen lines are erased until the terminating Enter is found.
← or 4	Moves the cursor one position left with Num Lock off. If the cursor advances beyond the left edge of the screen, the cursor will move to the right side of the screen on the previous line up.
→ or 6	Moves the cursor one position right with Num Lock off. If the cursor advances beyond the right edge of the screen, the cursor will move to the left side of the screen on the next line down.
↑ or 8	Moves the cursor one position up with Num Lock off.
↓ or 2	Moves the cursor one position down with Num Lock off.
Shift-↓ or Ctrl-N	Moves the cursor to the end of the logical line. Characters typed from this position are added to the end of the line.

Table 4-1 Special Program Editor Keys (cont'd)

KEY(S)	FUNCTION
Ctrl-F	Moves the cursor right to the next *word*. A word is defined as a character or group of characters which begins with a letter or number. Words are separated by blanks or special characters. So, the next word will be the next letter or number to the right of the cursor which follows a blank or special character. For example, suppose we have the following line: LINE (L1,LOW2)-(MAX,48) ,3 , BF As you can see, the cursor is presently in the middle of the word LOW2. If we press Next Word (Shift Cursor Right), the cursor will move to the beginning of the next word, which is MAX: LINE (L1,LOW2)-(MAX,48) ,3 , BF If we press Next Word again, the cursor will move to the next word, which is the number 48: LINE (L1,LOW2)-(MAX,48) ,3 , BF
Shift-← or Ctrl-B	Moves the cursor left to the previous word. The previous word will be the letter or number to the left of the cursor which is preceded by a blank or special character. For example, suppose we have: LINE (L1,LOW2)-(MAX,48) ,3 , BF_ If we press Previous Word (Shift Cursor Left), the cursor moves to the beginning of the word BF: LINE (L1,LOW2)-(MAX,48) ,3 , BF When we press Previous Word again, the cursor moves to the previous word, which is the number 3: LINE (L1,LOW2)-(MAX,48) ,3 , BF And if we press it twice more, the cursor will back up first to the number 48, then to the word MAX: LINE (L1,LOW2)-(MAX,48) ,3 , BF

Table 4-1 Special Program Editor Keys (cont'd)

KEY(S)	FUNCTION
Shift-Print	This is a special key that causes a copy of what is on the screen to be printed on the printer (LPT1:). So, if you ever need a hard (printed) copy of what is currently being displayed, just press and hold one of the Shift keys, and press the PRTSC key. (Note: Characters which are unrecognizable by the printer are printed as blanks.)
Ins or Ctrl-R	Sets insert mode. If insert mode is off, then pressing this key will turn it on. If insert mode is already on, then you will turn it off when you press this key. When you're in insert mode, the cursor covers the lower half of the character position. When insert mode is on, characters above and following the cursor move to the right as typed characters are inserted at the current cursor position. After each keystroke, the cursor moves one position to the right. Line folding occurs; that is, as characters advance off the right side of the screen they return on the left on a subsequent line. When insert mode is off, any characters typed replace existing characters on the line. Besides pressing the Ins key again, insert mode will also be turned off when you press any of the cursor movement keys or the Enter key.
Del	Deletes the character at the current cursor position. All characters to the right of the deleted character move one position left to fill in the empty space. Line folding occurs; that is, if a logical line extends beyond one physical line, characters on subsequent lines move left one position to fill in the previous space, and the character in the first column of each subsequent line moves up to the end of the preceding line.

Table 4-1 Special Program Editor Keys (cont'd)

KEY(S)	FUNCTION
Backspace or Ctrl-H	The Backspace key behaves somewhat differently from the Backspace key on a typewriter. It not only backspaces, it erases what you've typed as well. You should use the Cursor Left key to avoid erasing what you've typed.
Return or Enter or Ctrl-M	The key with the Return or Enter symbol on it is the carriage return key. You usually have to press this key to enter information into the computer. We will refer to it as the *Enter* key from now on. There are several important differences between this keyboard and a regular typewriter, however.
Clear Home or Ctrl-K	Moves the cursor to the upper left-hand corner of the screen.
Ctrl-Clear Home or Ctrl-L	Clears the screen and positions the cursor in the upper left-hand corner of the screen.
Ctrl-*	Ctrl-* serves as an on-off switch for any text sent to the screen to be also sent to your system printer. Press and hold the Ctrl key. Press the * key and then release both keys to print the display on the printer. Press and release both keys again to stop printing display output on the printer.
ESC	When pressed anywhere in the line, erases the entire logical line from the screen. The line is not passed to BASIC for processing. If it is a program line, it is not erased from the program in memory.
⇆TAB or Ctrl-I	Moves the cursor to the next tab stop. Tab stops occur every eight character positions; that is, at positions 1, 9, 17, etc. When insert mode is off, pressing the Tab key moves the cursor over characters until it reaches the next tab stop.

Table 4-1 Special Program Editor Keys (cont'd)

KEY(S)	FUNCTION
	For example, suppose we have the following line: 10 REM this is a remark If we press the Tab key, the cursor will move to the ninth position as shown: 10 REM this is a remark If we press the Tab key again, the cursor moves to the 17th position on the line: 10 REM this is a_remark When insert mode is on, pressing the Tab key inserts blanks from the current cursor position to the next tab stop. Line folding occurs as explained under Ins. For example, suppose we have this line: 10 REM this is a remark If we press the Ins key or Ctrl-R and then the Tab key, blanks are inserted up to position 17: 10 REM th is a remark
Ctrl-Return or Ctrl-J	This key extends logical lines to new physical lines. A new blank line is added to the last line in the current logical line.
Ctrl-T	This key is used to display on/off the soft keys. This key is alternating switch.

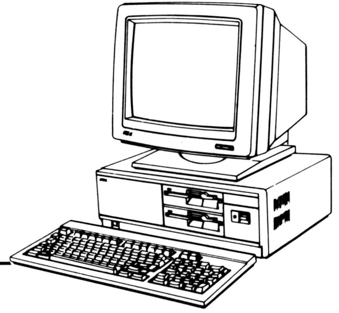
4.2.3 Logical Line Definition with INPUT

Normally, a logical line consists of all the characters on each of the physical lines that make up the logical line. During execution of an INPUT or LINE INPUT statement, however, this definition is modified slightly to allow for forms input. When either of these statements is executed, the logical line is restricted to characters actually typed or passed over by the cursor. The insert and delete modes move only characters that are within that logical line, and delete mode will decrement the size of the line.

Insert mode increments the logical line except when the characters moved will write over non-blank characters that are on the same physical line but not part of the logical line. In this case, the non-blank characters not part of the logical line are preserved and the characters at the end of the logical line are thrown out. This preserves labels that existed prior to the INPUT statement.

4.2.4 Editing Lines with Syntax Errors

When a syntax error is encountered during program execution, SLE GW-BASIC prints the line containing the error and enters direct mode. You can correct the error, enter the change, and reexecute the program. When a line is modified, all files are closed, and all variables are lost. Thus, if the user wishes to examine the contents of variables just before the syntax error was encountered, the user should print the values before modifying the program line. Alternative ways to get to direct mode without erasing variable values or closing files are the STOP and END commands.



Chapter 5

Working with Files and Devices

This chapter discusses the way files and devices are used and addressed in SLE GW-BASIC, and the way information is input and output through the system.

5.1 DEFAULT DEVICE

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

5.2 DEVICE-INDEPENDENT INPUT/OUTPUT

SLE GW-BASIC provides device-independent input/output that permits flexible approaches to data processing. Using device independent I/O means that the syntax for access is the same for any device.

The following statements, commands, and functions support device-independent I/O (see individual descriptions in Chapter 7):

BLOAD
BSAVE
CHAIN
CLOSE
EOF
GET
INPUT
INPUT$
LINE INPUT
LIST
LOAD
LOC
LOF
MERGE
OPEN
POS
PRINT
PRINT USING
PUT
RUN
SAVE
WIDTH
WRITE

5.3 FILENAMES AND PATHS

SLE GW-BASIC uses SLE MS-DOS's enhanced directory structure, allowing files to be accessed through their pathname.

5.3.1 Filename Specifications

File specifications follow SLE MS-DOS naming conventions. All filespecs may begin with a device specification such as A: or B: or COM1: or LPT1:. If no device is specified, the current drive is assumed. The default extension .BAS is appended to filenames used in LOAD, SAVE, MERGE and RUN <filename> commands, if no period (.) appears in the filespec and if the filename is less than nine characters long.

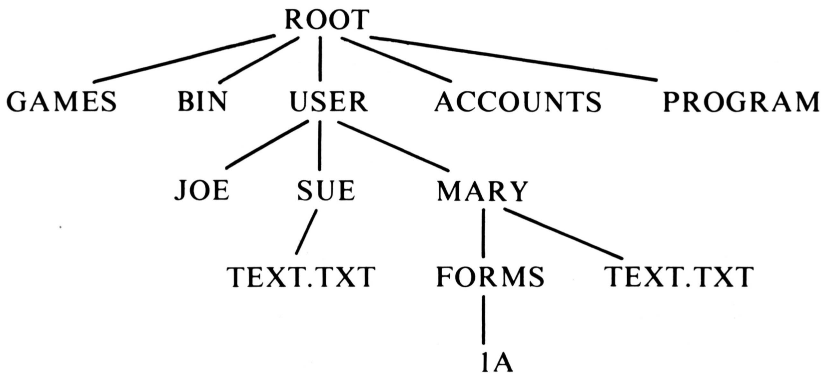
Examples:

```
RUN "NEWFILE.BAS"
RUN "A:NEWFILE.BAS"
RUN "KYBD:NEWFILE.BAS"
SAVE "NEWFILE" (file is saved with .BAS extension on default
               device)
```

5.3.2 Pathnames

A pathname is a sequence of directory names followed by a simple filename, each separated from the previous one by a backslash (\), and no longer than 128 characters. If a device is specified, it must be specified at the beginning of the pathname. A simple filename is a sequence of characters that can optionally be preceded by a drive designation, be devoid of backslashes, and be optionally followed by an extension.

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



A Sample Hierarchical Directory Structure

In the structure shown above, directories are in all upper-case letters. The two entries named TEXT.TXT and the entry named 1A are files.

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

The pathname of Sue's TEXT.TXT file is \USER\SUE\TEXT.TXT.

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

\	Indicates the root directory.
\PROGRAMS	Sample directory under the root directory containing program files.
\USER\MARY\FORMS\1A	A typical full pathname. This one happens to be a file named 1A in the directory named FORMS belonging to the subdirectory of USER named MARY.

USER\SUE	A relative pathname; it names the file or directory SUE in subdirectory USER of the working directory. If the working directory is the root (\), (\), it names \USER\SUE.
TEXT.TXT	Name of a file or directory in the working directory.

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

. SLE MS-DOS uses this shorthand notation to indicate the name of the working directory in all hierarchical directory listings. SLE MS-DOS automatically creates this entry when a directory is made.

.. The shorthand name of the working directory's parent directory. If you type:

```
DIR ..
```

then SLE MS-DOS will list the files in the parent directory of your working directory.

If you type:

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

then SLE MS-DOS will list the files in the parent's PARENT directory.

5.3.3 Working with Pathnames in BASIC

Not only can BASIC provide the ability to access files from other directories using pathname approaches, but it can also be used to create, change, and remove paths, using the BASIC commands MKDIR, CHDIR, and RMDIR.

The BASIC statement MKDIR "ACCOUNTS" would create a new directory, ACCOUNTS, in the working directory of the current drive.

The BASIC statement CHDIR "B:EXPENSES" would change the current directory on B: to EXPENSES.

The BASIC statement RMDIR "CLIENTS" would delete an existing directory, CLIENTS, as long as that directory was empty of all files with the exception of "." and "..".

For further information on handling paths in BASIC, see CHDIR, ENVIRON, ENVIRON$, MKDIR, and RMDIR Statements in Chapter 7.

5.4 RE-DIRECTION OF STANDARD INPUT AND STANDARD OUTPUT

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

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

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

Rules:

1. When re-directed, all INPUT, LINE INPUT, INPUT$, and INKEY$ statements will read from the input file.
2. If the program does not specify a file number in a PRINT statement, that output is directed to the declared output file as well as the standard output device, the screen.
3. Error messages go to standard output.
4. File input from "KYBD:" still reads from the keyboard.
5. File output to "SCRN:" still outputs to the screen.
6. BASIC will continue to trap keys from the keyboard when the ON KEY(n) statement is used.
7. The printer echo key will not cause LPT1: echoing if Standard Output has been re-directed.
8. Typing Control-Break/Stop will cause BASIC to close any open files, issue the message "Break in line <line_number<" to standard output, and exit BASIC.

9. When input is redirected, BASIC will continue to read from this source until an end-of-file character is detected. This condition may be tested with the EOF function. If the file is not terminated by a Control-Z, or a BASIC input statement tries to read past end-of-file, then any open files are closed, the message "Read past end" is written to standard output, and BASIC terminates.

Examples:

```
BASIC MYPROG >DATA.OUT
```

Data read by INPUT and LINE INPUT will continue to come from the keyboard. Data output by PRINT will go into the file DATA.OUT.

```
BASIC MYPROG <DATA.IN
```

Data read by INPUT and LINE INPUT will come from DATA.IN. Data output by PRINT will continue to go to the screen.

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

Data read by INPUT and LINE INPUT will now come from the file MYINPUT.DAT and data output by PRINT will go into MYOUTPUT.DAT.

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

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

5.5 HANDLING FILES

File I/O procedures for the beginning BASIC user are examined in this section. If you are new to BASIC, or if you are encountering file-related errors, read through these procedures and program examples to make sure you are using all the file statements correctly.

5.5.1 Program File Commands

The following is a review of the commands and statements used in program file manipulation. All file specifications may include the device and pathname.

SAVE <filespec> {[,A:P]}	Writes the program that currently resides in memory to the specified file. Option A writes the program as a series of ASCII characters. With option P, BASIC will encode the file in a read-protected format.
LOAD <filespec> [,R]	Loads the program from file into memory. The optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before loading. If R is included, however, open data files are kept open. Thus, programs can be chained or loaded in sections and access the same data files. (LOAD FILESPEC>,R and RUN FILESPEC),R are equivalent.)
RUN <filespec> [,R]	Loads the program from file into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN<filespec>,R and LOAD <filespec>,R are equivalent.)
MERGE <filespec>	Loads the program from file memory out but does not delete the current contents of memory. The program line numbers in the file are merged with the line numbers in memory. If two lines have the same number, only the line from the file program is saved. After a MERGE command is executed, the "merged" program resides in memory, and BASIC returns to command level. In order to successfully MERGE a program, the <filespec> must have been saved in ASCII format.

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

Where <line number expression> is the line number in the new program at which the program is to start execution. Passes control to the named program, and passes the use of the variables and their current values to the new program. The user may choose to start the new program on a specified line, delete some lines, or transfer the values of only some of the variables.

KILL <filespec>

Deletes the file from the disk. <filespec> can be a program file or a sequential or random access data file.

NAME <old filespec>
AS <new filespec>

Changes the name of a file. NAME AS <filespec> can be used with program files, random access files, or sequential files.

5.5.2 Protecting Program Files

If you wish to have a program saved in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

5.6 DATA FILES: SEQUENTIAL AND RANDOM ACCESS I/O

There are two types of disk data files that can be created and accessed by a BASIC program: sequential files and random access files.

5.6.1 Sequential Files

Sequential files are easier to create than random access files, but are limited in flexibility and speed when it comes to locating data. The data written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order sent. The data is read back sequentially, one item after another.

The following statements and functions are used with sequential data files in sequential order.

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

5.6.1.1 CREATING A SEQUENTIAL FILE

Program 1 is a short program that creates a sequential file, "DATA," from information you input at the keyboard.

Program 1—Create a Sequential Data File

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$ = "DONE" THEN END
30 INPUT "DEPARTMENT";DEPT$
40 INPUT "DATE HIRED";HIREDATE$
50 PRINT#1,N$;",";DEPT$;",";HIREDATE$
60 PRINT
70 GOTO 20
RUN
NAME? SAMUEL GOLDWYN
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72
NAME? MARVIN HARRIS
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65
NAME? DEXTER HORTON
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/81
NAME? STEVEN SISYPHUS
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/81
NAME? etc.
```

As illustrated in Program 1, the following program steps are required to create a sequential file and access the data in it:

1. OPEN the file in "O" mode.
2. Write data to the file using the PRINT# statement. (WRITE# can be used instead.)
3. To access the data in the file, you must CLOSE the file and reopen it in "I" mode.
4. Use the INPUT# statement to read data from the sequential file into the program.

5.6.1.2 READING DATA FROM A SEQUENTIAL FILE

Now look at Program 2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1981.

Program 2—Accessing a Sequential File

```
10 OPEN"I",#1,"DATA"
20 INPUT#1,N$,DEPT$,HIREDATE$
30 IF RIGHT$(HIREDATE$,2) = "81" THEN PRINT N$
40 GOTO 20
RUN

DEXTER HORTON
STEVEN SISYPHUS
Input past end in 20
```

Program 2 reads, sequentially, every item in the file, and prints the names of employees hired in 1981. When all the data has been read, line 20 causes an INPUT PAST END error. To avoid this error, use the WHILE... WEND control structure, which uses the EOF function to test for the end-of-file. The revised program looks like:

```
10 OPEN"I",#1,"DATA"
15 WHILE NOT EOF(1)
20      INPUT#1,N$,DEPT$,HIREDATE$
30      IF RIGHT$(HIREDATE$,2) = "81" THEN PRINT N$
40 WEND
```

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

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

could be used to write numeric data to the file without explicit delimiters. The commas at the end of the format string separate the items in the disk file.

If the user wants commas to appear in the file as delimiters between variables, the WRITE statement can be used. For example, the statement

```
WRITE 1, A, B$
```

could be used to write these two variables to the file with commas delimiting them.

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

5.6.1.3 ADDING DATA TO A SEQUENTIAL FILE

If you have a sequential file residing on disk and want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in the output ("O") mode, you destroy its current contents.

Instead, use the append ("A") mode. If the file doesn't already exist, the open statement will work exactly as it would if output ("O") mode had been specified.

The following procedure can be used to add data to an existing file called "FOLKS".

Program 3—Adding Data to a Sequential File

```
110 OPEN "A",#1,"FOLKS"
120 REM ADD NEW ENTRIES TO FILE
130 INPUT "NAME";N$
140 IF N$=" " THEN 201'CARRIAGE RETURN EXITS INPUT LOOP
150 LINE INPUT "ADDRESS? ";ADDR$
160 LINE INPUT "BIRTHDAY? ";BIRTHDATE$
170 PRINT#1,N$
180 PRINT#1,ADDR$
190 PRINT#1,BIRTHDATE$
200 GOTO 120
210 CLOSE 1
```

5.6.2 Random Access Files

Creating and accessing random access files requires more program steps than creating and accessing sequential files. However, there are advantages to using random access files. One advantage is that random access files require less room on the disk, since BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage of using random access files is that data can be accessed randomly, i.e., anywhere on the disk. It is not necessary to read through all the information from the beginning of the file, as with sequential files. This is possible because the information is stored and accessed in distinct units called records, each of which is numbered.

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

STATEMENTS	FUNCTIONS
OPEN	CVD
FIELD	CVI
GET	CVS
LOC	MKS$
LOF	MKD$
LSET	MKI$
RSET	
PUT	
CLOSE	

5.6.2.1 CREATING A RANDOM ACCESS FILE

Program 4—Create a Random File

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

5.6.2.2 ACCESSING A RANDOM ACCESS FILE

Program 5 accesses the random access file "FILE" that was created in Program 4. By entering a three-digit code at the keyboard terminal, the information associated with that code is read from the file and displayed.

Program 5—Access a Random File

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

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

1. OPEN the file in "R" mode.

   Example:

   ```
   OPEN "R", 1,"FILE",32
   ```

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

   Example:

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

   NOTE

   In a program that performs both input and output on the same random access file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random access buffer.

   Example:

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

4. The data in the buffer can now be accessed by the program. Numeric values that were converted to strings by the MKS$, MKD$ or MKI$ statements must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values. The MKI$ and CVI processes mirror each other, the former converting a number into a format for storage in random files, the latter converting the random file storage into a format usable by the program.

   Example:

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

The LOC function, when used with random access files, returns the "current record number." The current record number is the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1) > 50 THEN END
```

ends program execution if the current record number in file#1 is greater than 50.

5.6.2.3 RANDOM FILE OPERATIONS

Program 6 is an inventory program that illustrates random file access.

As illustrated by Program 4, the following program steps are required to create a random access file.

1. OPEN the file for random access ("R" mode). The following example specifies a record length of 32 bytes. If the record length is not specified, the default is 128 bytes unless it was set to another value with the /I/S: switches when invoking BASIC (see Chapter 2 for details).

   Example:

   ```
   OPEN "R", 1,"FILE",32
   ```

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

   Example:

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

3. Use LSET to move the data into the random access buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ to make a single precision value into a string, and MKD$ to make a double precision value into a string.

   Example:

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

4. Write the data from the buffer to the disk using the PUT statement.

   Example:

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

Program 4 takes information that is input at the terminal and writes it to a random access file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a fielded string variable in an INPUT or LET statement. Doing so causes that variable to be redeclared; BASIC will no longer associate that variable with the file buffer, but with the new program variable.

Program 6—Inventory

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

```
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";ADDITIONAL%
520 Q%=CVI(Q$)+ADDITIONAL%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";LESS%
610 Q%=CVI(Q$)
620 IF (Q%-LESS%)<O THEN PRINT "ONLY";Q%;"
    IN STOCK":GOTO 600
630 Q%=Q%-LESS%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
    " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710   GET#1,I
720   IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
      CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART %
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART
    NUMBER": GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";CONFIRM$:
    IF CONFIRM$< >"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940   PUT#1,I
950 NEXT I
960 RETURN
```

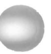
In this program, the record number is used as the part number. It is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

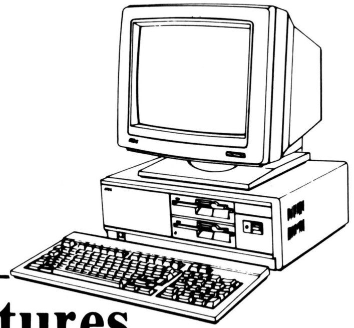
Lines 130-220 display the various inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

5.7 BASIC AND CHILD PROCESSES

Through the use of the SHELL statement, SLE GW-BASIC is able to use one of the most powerful features of SLE MS-DOS: the ability to create child processes. SHELL enables the user to run part of a BASIC program, temporarily exit to SLE MS-DOS to perform a specified function, and return to the BASIC program at the statement after the SHELL statement to proceed with the rest of the program.

BASIC will produce a child program when it uses the SHELL statement. It is not possible for BASIC to totally protect itself from its children. When a SHELL statement is executed, many things may be going on. For example, files may be OPEN and devices may be in use. It is advisable for programmers to thoroughly read about the SHELL Statement, Chapter 7, before using this powerful statement.





Chapter 6

Using Advanced Features

6.1 ASSEMBLY LANGUAGE SUBROUTINES

You may call assembly language subroutines from your SLE GW-BASIC program with the USR function or the CALL or CALLS statement.

It is recommended that you use the CALL or CALLS statement for interfacing 8086 machine language programs with SLE GW-BASIC. These statements are more readable and can pass multiple arguments. In addition, the CALL statement is compatible with more languages than its alternative, the USR function.

6.1.1 Memory Allocation

Memory space must be set aside for an assembly language subroutine before it can be loaded. To do so, use the /M: switch during start-up. The /M: switch sets the highest memory location to be used by SLE GW-BASIC.

In addition to the SLE GW-BASIC code area, SLE GW-BASIC uses up to 64K of memory beginning at its data (DS) segment.

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

The assembly language subroutine can be loaded into memory in several ways, the most simple being to use the BLOAD command (see BLOAD Command, Section 7.5). Also, the user could SHELL a program that exits, but stays resident, leaving the linked, relocated image in memory. As a third choice, the user could execute a program that exits but stays resident, and then run BASIC.

The following guidelines must be observed if you choose to BLOAD, or read and poke, an EXE file into memory:

1. Make sure the subroutines do not contain any long references, address offsets that exceed 64K, or that take the user out of the code segment. These long references require handling by the EXE loader.
2. Skip over the first 512 bytes (the header) of the linker's output file (EXE), then read in the rest of the file.

6.1.2 Internal Representation

The following section describes the internal representation of numbers in SLE GW-BASIC. Knowledge of these arrangements is critical for many assembly language programming routines.

Single Precision - 24 bit mantissa

0	1	2	3
loman		S himan	exp

where loman = the low mantissa
S = the sign
himan = the high mantissa
exp = the exponent
man = himan:...:loman

- If <exp> = 0, then <number> = 0.
- If <exp> <> 0, then the mantissa is normalized and

  <number> = <sgn> * .1 <man> * 2 ** (<exp> -80h)

  That is, in single precision (hex notation - bytes low to high)

  00000080 = .5
  00008080 = -.5

Double Precision - 56 bit mantissa

0	1	2	3	4	5	6	7
loman						S himan	exp

6.1.3 CALL Statement

The CALL statement is the recommended way of interfacing 8086 machine language subroutines with SLE GW-BASIC. Do not use the USR function unless you are running previously written subroutines that already contain USR functions.

The syntax of the CALL statement is:

> CALL <variable name> [(<argument list>)]
>
> where <variable name> contains the offset into the current segment that is the starting point in memory of the subroutine being called. The current segment is either the default, or that which has been defined by a DEF SEG statement.
>
> <argument list> contains the variables or constants, separated by commas, that are to be passed to the subroutine.

Invoking the CALL statement causes the following to occur:

1. For each argument in the argument list, the two-byte offset of the argument's location within the BASIC segment is pushed onto the stack.
2. Control is transferred to the subroutine with an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in <variable name>.

Figures 6-1 and 6-2 illustrate the state of the stack at the time the CALL statement is executed, and the condition of the stack during execution of the called subroutine, respectively.

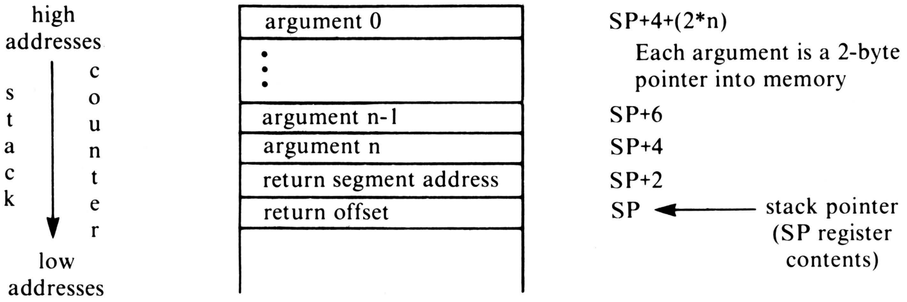


Figure 6-1 Stack Layout When CALL Statement Is Activated

After the CALL statement has been activated, the subroutine has control. Arguments may be referenced by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP.

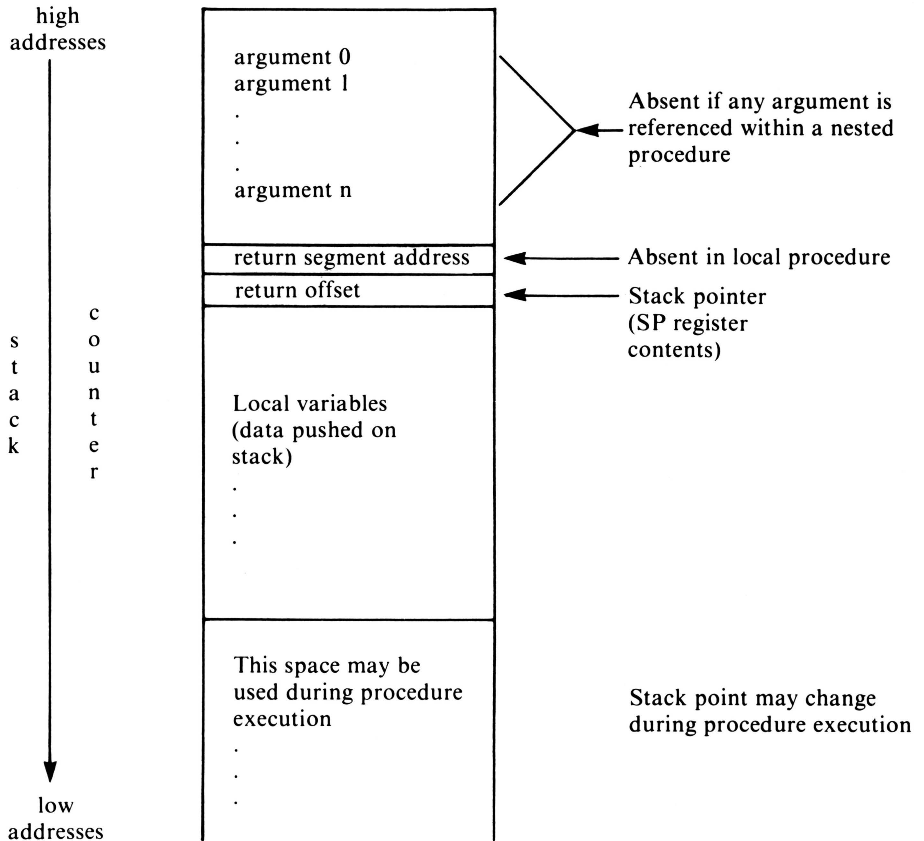


Figure 6-2 Stack Layout During Execution of a CALL Statement

Observe the following rules when coding a subroutine:

1. The called routine must preserve segment registers DS, ES, SS, and the base pointer (BP). If interrupts are disabled in the routine, they must be enabled before exiting. The stack must be cleaned up on exit.

2. The called program must know the number and length of the arguments passed. The following routine shows an easy way to reference arguments:

```
PUSH     BP
MOV      BP,SP
ADD      BP, (2*number of arguments)+4
```

   Then:

   argument 0 is at BP
   argument 1 is at BP-2
   argument n is at (BP-2)*n

   (number of arguments = n+1)

3. Variables may be allocated either in the code segment or on the stack. Be careful not to modify the return segment and offset stored on the stack.

4. The called subroutine must clean up the stack. A preferred way to do this is to perform a RET <n> statement (where <n> is two times the number of arguments in the argument list) to adjust the stack to the start of the calling sequence.

5. Values are returned to SLE GW-BASIC by including in the argument list the name of the variable that will receive the result. The internal format for numbers in SLE GW-BASIC is discussed in "Internal Representation," Section 6.1.2.

6. If the argument is a string, the argument's offset points to 3 bytes which, as a unit, are called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

WARNING

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. For example, use

20 A$ = "BASIC",""

This will force the string literal to be copied into string space. Then the string may be modified without affecting the program.

7. The contents of a string may be altered by user routines, but the descriptor must not be changed.

   Do not write past the end-of-string. SLE GW-BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

8. Data areas needed by the routine must be allocated either in the CODE segment of the user routine or on the stack. It is not possible to declare a separate data area in the user assembler routine.

Example of CALL statement:

```
100 DEF SEG=&H8000
110 F00=&H7FA
120 CALL F00(A,B$,C)
 .
 .
 .
```

Line 100 sets the segment to 8000 Hex. The value of variable F00 is added into the address as the low word after the DEF SEG value is left shifted 4 bits. Here, the long call to F00 will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

The following sequence in 8086 assembly language demonstrates access to the arguments passed. The returned result is stored in the variable 'C'.

```
PUSH    BP              ;Set up pointer to arguments
MOV     BP,SP
ADD     BP,(4+2*3)
MOV     BX,[BP-2]       ;Get address of B$ descriptor
MOV     CL,[BX]         ;Get length of B$ in CL
MOV     DX,1[BX]        ;Get addr of B$ text in DX
         .
         .
         .
MOV     SI,[BP]         ;Get address of 'A' in SI
MOV     DI[BP-4]        ;Get pointer to 'C' in DI
MOVS    WORD            ;Store variable 'A' in 'C'
POP     BP              ;Restore pointer
RET     6               ;Restore stack, return
```

NOTE

The called program must know the variable type for the numeric arguments passed. In the previous example, the instruction

MOVS WORD

will copy only two bytes. This is fine if variables A and C are integer. You would have to copy four bytes if the variables were single precision format and copy 8 bytes if they were double precision.

6.1.4 CALLS Statement

The CALLS statement should be used to access subroutines that were written using MS-FORTRAN calling conventions. CALLS works just like CALL, except that with CALLS the arguments are passed as segmented addresses, rather than as unsegmented addresses.

Because MS-FORTRAN routines need to know the segment value for each argument passed, the segment is pushed and then the offset is also pushed. For each argument, four bytes are pushed rather than 2, as in the CALL statement. Therefore, if your assembler routine uses the CALLS statement, n in the RET <n> statement is four times the number of arguments.

6.1.5 USR Function

Although using the CALL statement is the recommended way of calling assembly language subroutines, the USR function is also available for this purpose. This ensures compatibility with older programs that contain USR functions.

> USR[<digit>] [(<argument>)]
>
> where <digit> is from 0 to 9. <digit> specifies which USR routine is being called. If <digit> is omitted, USR0 is assumed.
>
> <argument> is any numeric or string expression. Arguments are discussed in detail in the following paragraphs.

In the SLE GW-BASIC Interpreter, a DEF SEG statement must be executed prior to a USR function call to assure that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine.

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

6.2 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 specific 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 a communications port (ON COM), detection of certain keystrokes (ON KEY), and time passage (ON TIMER).

This section gives an overview of event trapping. For more details on individual statements, see Chapter 7.

Event trapping is controlled by the following statements:

<event specifier> ON to turn on trapping
<event specifier> OFF to turn off trapping
<event specifier> STOP to temporarily turn off trapping

where <event specifier> is one of the following:

COM (1) where 1 is the number of the communications channel.

Typically, the COM trap routine will read an entire message from the COM port before returning. We do not recommend using the COM trap for single character messages because at high baud rates the overhead of trapping and reading for each character may allow the interrupt buffer for COM to overflow.

KEY (n) where n is a trappable key number. Trappable keys are numbered 1 through 20.

Note that KEY (n) ON is not the same statement as KEY ON. KEY (n) ON sets an event trap for the specified key. KEY ON displays the values of the function keys on the twenty-fifth line of the screen (see Sections 7.66 and 7.67).

When the GW-BASIC Interpreter is in direct mode function keys maintain their standard meanings.

When a key is trapped, that occurrence of the key is destroyed. Therefore, you cannot subsequently use the INPUT or INKEY$ statements to find out which key caused the trap. So if you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

TIMER — ON TIMER(n), where (n) is a numeric expression representing a number of seconds. The ON TIMER statement can be used to perform background tasks at defined intervals.

6.2.1 ON GOSUB Statement

The ON GOSUB statement sets up a line number for the specified event trap. The format is:

ON <event specifier> GOSUB <line number>

A <line number> of zero disables trapping for that event.

When an event is ON and if a non-zero line number has been specified in the ON GOSUB statement, every time SLE GW-BASIC starts a new statement it will check to see if the specified event has occurred (e.g., the lightpen has been struck or a COM character has come in). When an event is OFF, no trapping takes place, and the event is not remembered even if it takes place.

When an event is stopped (<event specifier> STOP), no trapping takes place, but the occurrence of an event is remembered so that an immediate trap will take place when an <event specifier> ON statement is executed.

When a trap is made for a particular event, the trap automatically causes a STOP on that event, so recursive traps can never occur. A return from the trap routine automatically executes an ON statement unless an explicit OFF has been performed inside the trap routine.

Note that once an error trap takes place, all trapping is automatically disabled. In addition, event trapping will never occur when SLE GW-BASIC is not executing a program.

6.2.2 RETURN Statement

When an event trap is in effect, a GOSUB statement will be executed as soon as the specified event occurs. For example, the statement

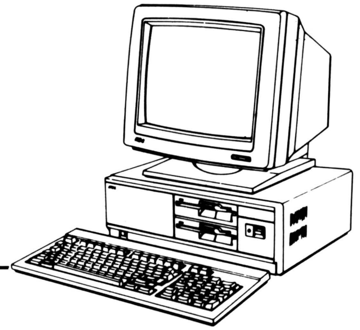
```
ON KEY(1) GOSUB 1000
```

specifies that the program go to line 1000 as soon as key "1" is pressed. If a simple RETURN statement is executed at the end of this subroutine, program control will return to the statement following the one where the trap occurred. When the RETURN statement is executed, its corresponding GOSUB return address is cancelled.

SLE GW-BASIC includes the RETURN <line number> enhancement, which lets processing resume at a definable line. Normally, the program returns to the statement immediately following the GOSUB statement when the RETURN statement is encountered. However, RETURN <line number> enables the user to specify another line. If not used with care, however, this capability may cause problems. Assume, for example, that your program contains:

```
10 ON KEY(1) GOSUB 1000
20 KEY(1) ON
30 FOR I = 1 TO 10
40 PRINT I
50 NEXT I
60 REM NEXT PROGRAM LINE
200 REM PROGRAM RESUMES HERE
1000 'FIRST LINE OF SUBROUTINE
 .
 .
 .
1050 RETURN 200
```

If the pen is activated while the FOR/NEXT loop is executing, the subroutine will be performed, but program control will return to line 200 instead of completing the FOR/NEXT loop. The original GOSUB entry will be cancelled by the RETURN statement, and any other GOSUB, WHILE, or FOR that was active at the time of the trap will remain active. But the current FOR context will also remain active, and a "FOR without NEXT" error may result.



Chapter 7

Basic Commands, Functions and Statements

7.1 ABS FUNCTION

Syntax:

```
ABS(X)
```

Purpose:

To return the absolute value of the expression X.

Example:

```
PRINT ABS(7*(-5))
will yield
35
```

7.2 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 A 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 7.12, for details on ASCII-to-string conversion.

7.3 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 default evaluation of ATN is performed in single precision. This may be overridden if the /D switch is used when invoking SLE GW-BASIC.

Example:

```
10 LET X = 3
20 PRINT ATN(X)
```

will yield

1.249046

7.4 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 the Break or Ctrl-C key. The line in which the Break or Ctrl-C key is typed will not be saved. After the Break or Ctrl-C key is typed, SLE GW-BASIC returns to command level.

Example:

AUTO 100,50	Generates line numbers 100, 150, 200 . . . .
AUTO	Generates line numbers 10, 20, 30, 40 . . . .

7.5 BLOAD COMMAND

Syntax:

BLOAD <filespec> [,<offset>]

The device designation portion of the filespec is optional. The filename, not including the device designation, 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 any input device.

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 Command, in Section 7.6, 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 SLE 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 SLE GW-BASIC or the operating system.

Example:

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

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

7.6 BSAVE COMMAND

Syntax:

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

The device designation portion of the filespec is optional. The filename, not including the device specification, must 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 transfer the contents of the specified area of memory to any output device.

Remarks:

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

The BSAVE command 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.

A DEF SEG statement may be executed before the BSAVE. The last known DEF SEG address will be used for the save.

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.BAS

7.7 CALL STATEMENT

Syntax:

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

where <numeric variable name> contains an address that is the starting point in memory of the subroutine. <numeric 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 7.148.)

See Section 6.1.3, CALL Statement, for a detailed discussion of calling sequences.

Example:

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

7.8 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 the FORTRAN calling convention. All FORTRAN parameters are call-by-reference segmented addresses.

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

7.9 CDBL FUNCTION

Syntax:

CDBL(X)

Purpose:

To convert X to a double precision number.

Example:

```
10 LET PI = 22/7
20 PRINT PI,CDBL(PI)
```

will yield

3.142857 3.142857074737549

7.10 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 a string expression containing a name that conforms to SLE MS-DOS rules for disk filenames or SLE GW-BASIC rules for device specifications.

<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 7.20 for information about COMMON.

If the ALL option is used and <line number exp> is not, a comma must hold the place of <line number exp>. For example, CHAIN "NEXT-PROG",,ALL is correct; CHAIN "NEXTPROG",ALL is incorrect. In the latter case, SLE 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 SLE GW-BASIC program as an overlay. That is, the current program and the called program are merged (see MERGE Command, Section 7.85). 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.

Examples:

CHAIN is used in different ways in the two examples below. In the first, the two string arrays are dimensioned, and declared as common variables. When the program gets to line 90, it chains to the other program, which loads the B$s. At line 90 of PROG2, control chains back to the first program, but line 100 is delineated, and so the first program executes from that line. This process can be observed through the descriptive text that prints as the programs execute.

Example 1:

(PROG1.BAS)

```
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$(1)=""
80 B$(2) =""
90 CHAIN "PROG2"
100 PRINT
110 PRINT B$(1)
120 PRINT
130 PRINT B$(2)
140 PRINT
150 END
```

(PROG2.BAS)

```
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
60 PRINT A$(1);A$(2)
70 B$(1)="NOTE HOW THE OPTION OF SPECIFYING
      A STARTING LINE NUMBER"
80 B$(2)="WHEN CHAINING AVOIDS THE DIMENSION
      STATEMENT IN 'PROG1'."
90 CHAIN "PROG1",100
100 END
```

In the second example, the MERGE, ALL, and DELETE options are illustrated. After A$ is loaded in the first program, control chains to line 1010 of the second. At the second program's line 1040, it chains to line 1010 of the third program, keeping all variables and deleting all the second program's lines. Control passes to the third program. This process can be observed through the descriptive text that prints as the programs execute.

Example 2:

```
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 snared 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.

7.11 CHDIR STATEMENT

Syntax:

CHDIR PATHNAME

Purpose:

To change the current operating directory.

Remarks:

PATHNAME is a string specifying the name of the directory which is to be the current directory. CHDIR works exactly like the SLE MS-DOS command CHDIR. The PATHNAME must be a string of less than 128 characters.

Example:

CHDIR "SALES"

This makes SALES the current directory.

CHDIR "B:USERS"

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

Also see the MKDIR and RMDIR statements.

7.12 CHR$ FUNCTION

Syntax:

CHR$(I)

Purpose:

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

Remarks:

CHR$ is commonly used to send a special character to the screen or printer. For instance, 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 7.2, for details on ASCII-to-numeric conversion.

7.13 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.

7.14 CIRCLE STATEMENT

Syntax:

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

The [STEP] option makes the specified center and ycenter coordinates relative to the "most recent point," instead of absolute, mapped coordinates.

<xcenter> is the x coordinate for the center of the circle.

<ycenter> is the y coordinate for the center of the circle.

<radius> is the radius of the circle in the current logical coordinate system.

<color> is the numeric symbol for the color desired (see COLOR Statement, Section 7.18). The default color is the foreground color.

<start> and <end> are the start and end angles in radians. The range is -2 * pi through 2 * pi. These angles allow the user to specify where an ellipse will begin and end. If the start or end angle is negative, the ellipse will be connected to the center point with a line, and the angles will be treated as if they were positive. Note that this is different from adding 2 * pi. The start angle may be less than the end angle.

<aspect> is the aspect ratio, i.e., the ratio of the x radius to the y radius. Default ratios depend on the machine that is being used. When default ratios are specified for the corresponding screen mode, a round circle is drawn.

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

Purpose:

To draw an ellipse or circle with the specified center and radius.

Remarks:

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

It is not an error to supply coordinates that are outside the screen or viewport.

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

STEP (<xoffset> <yoffset>)

For example, if the most recent point referenced were (10,10), STEP (10,5) would reference a point offset 10 from the current x coordinate and offset 5 from the current y coordinate, that is, the point (20,15).

Example:

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

CIRCLE (200,200),50

and

CIRCLE STEP (100,150),50

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

7.15 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 SLE GW-BASIC.

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

NOTE

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.

Examples:

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

7.16 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 of a particular file and a file number terminates upon execution of a CLOSE statement. The file may then be reopened using the same or a different file number. Once a file is closed, that file's number may be used for any unopened file.

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

The SYSTEM, CLEAR, and END statements and the NEW and RESET commands always close all files automatically.

Example:

CLOSE #1,#2

7.17 CLS STATEMENT

Syntax:

CLS

Purpose:

Erase contents of entire current screen.

Remarks:

If the screen is in text mode, the active page (see "SCREEN Statement") is cleared to the background color.

If the screen is in graphics mode (medium or high resolution), the entire screen buffer is cleared to the background color.

The CLS statement also returns the cursor to the home position. In text mode, this means the cursor is located in the upper left-hand corner of the screen. In graphics mode, this means the "last referenced point" for future graphics statements is the point in the center of the screen ((160, 100) in medium resolution, (320,100) in high resolution).

Changing the screen mode or width by using the SCREEN or WIDTH statements also clears the screen. The screen may also be cleared by pressing Ctrl-Home.

Example:

```
10 COLOR 10,1
20 CLS
```

7.18 COLOR STATEMENT

Purpose:

Sets the colors for the foreground and background screen. The syntax of the COLOR statement depends on whether you are in the text mode or graphics mode, as set by the SCREEN statement.

The COLOR Statement in Text Mode:

Syntax:

COLOR [*foreground*]

Remarks:

foreground — is a numeric expression in the range 0 to 7 and 16 to 23, representing the character color.

If you have the Color Graphics Board, the following colors are allowed in *foreground*:

0 Black
1 Blue
2 Green
3 Cyan
4 Red
5 Magenta
6 Yellow
7 White

You can make the characters blink by setting *foreground* equal to 16 plus the number of the desired color. That is, a value of 16 to 23 causes blinking characters.

The COLOR Statement in Graphics Mode:

Syntax:

Graphics mode, high resolution only.

COLOR [*foreground*]

Graphics mode, medium resolution only.

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

Remarks:

foreground is an integer expression in the range 0 through 7.

background is an integer expression in the range 0 through 7. It specifies the background attribute.

palette is an integer expression. It selects one of two palettes of color.

In graphics mode, the COLOR statement sets a background color and chooses one of two palettes with four color attributes each (0-3). Color attribute 0 is always the current background. You can select one of three color attributes for the foreground color to be used with PSET, PRESET, LINE, CIRCLE, PAINT, VIEW, and DRAW. The colors selected when you choose each palette are as follows:

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

If *palette* is an even number, palette 0 is selected. This associates the colors green, red, and brown to the color attributes 1, 2, and 3.

If *palette* is an odd number, palette 1 (cyan/magenta/white) is selected.

Example:

This statement sets the background to blue and selects palette 0.

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

In the next example, the background stays blue, and palette 1 is selected.

```
10 COLOR ,1
```

7.19 COM STATEMENT

Syntax:

```
COM(1) ON
COM(1) OFF
COM(1) STOP
```

Purpose:

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

Remarks:

The COM(1) ON statement enables communications event trapping by an ON COM statement (see ON COM Statement, Section 7.93). While trapping is enabled, and if a non-zero line number is specified in the ON COM statement, SLE 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(1) OFF disables communications event trapping. If an event takes place, it is not remembered.

COM(1) STOP disables communications event trapping, but if an event occurs, it is remembered. If there is a subsequent COM(1) ON statement, the remembered event will be successfully trapped.

NOTE

For additional information on communications event trapping, see "Event Trapping," Section 6.2, and ON COM Statement, Section 7.93.

Example:

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

Enables event trapping of communications activity.

7.20 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. SLE 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
 .
 .
 .
```

7.21 CONT COMMAND

Syntax:

CONT

Purpose:

To continue program execution after a Break has been typed or a STOP 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 7.136.

7.22 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, unless the / D switch is specified when BASIC is invoked and either the argument that receives the value of the cosine is a double precision variable or (X) is specified a double precision number with the # sign.

Example:

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

will yield

```
1.842122
```

7.23 CSNG FUNCTION

Syntax:

CSNG(X)

Purpose:

To convert X to a single precision number.

Example:

```
10 A# = 975.3421115#
20 PRINT A#, CSNG(A#)
```

will yield

```
975.3421115        975.3421
```

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

7.24 CSRLIN FUNCTION

Syntax:

```
CSRLIN
```

CSRLIN returns the current line position.

Purpose:

To obtain the current line position of the cursor in a numeric variable.

Remarks:

To return the current column position, use the POS function (Section 7.107).

Example:

```
10 y = CSRLIN 'Record current line.
20 x = POS(0) 'Record current column.
30 LOCATE 24,1
40 PRINT "HELLO"
50 LOCATE x,y 'Restore position to old line and column
```

7.25 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 a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example:

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

See also MKI$, MKS$, MKD$ Functions, Section 7.89.

7.26 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 7.116.)

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 7.120).

Example:

See READ Statement, Section 7.116.

7.27 DATE$ STATEMENT

Syntax:

DATE$=<string expression>

<string expression> must be 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.

Remarks:

The year must be in the range 1980 to 2099. If you use only one digit for the month or day, a 0 (zero) is assumed in front of it. If you give only one digit for the year, a zero is appended to make it two digits. If you give only two digits for the year, if that is in the range 80 to 99, the year is assumed to be 19yy, else 20yy.

Example:

```
10 DATE$="07-01-1983"
```

The current date is set at July 1, 1983.

7.28 DATE$ FUNCTION

Syntax:

DATE$

Purpose:

To retrieve the current date. (To set the date, use the DATE$ statement, described in Section 7.27.)

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.

7.29 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.

7.30 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 3.3.1.)

If no type declaration statements are encountered, SLE 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.

7.31 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 SLE GW-BASIC data segment. This is the initial default value.

NOTE

DEF and SEG must be separated by a space. Otherwise, SLE 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 at B800 Hex
20 DEF SEG 'Restore segment to SLE GW-BASIC data segment
```

7.32 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)
 .
 .
 .
```

7.33 DELETE COMMAND

Syntax:

DELETE {[<line number>][-<line number>][<line number>-]}

Purpose:

To delete program lines.

Remarks:

SLE 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.

```
DELETE 40-
```

Deletes lines 40 through the end, inclusive.

```
DELETE .
```

Deletes current line.

7.34 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 0, unless otherwise specified with the OPTION BASE statement (see Section 7.100).

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

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.

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, it is good programming practice to put the required DIM statements at the beginning of a program, outside of any processing loops.

Example:

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

7.35 DRAW STATEMENT

Syntax:

DRAW <string expression>

where <string expression> is one of the subcommands described below in "Remarks."

Purpose:

To draw an object defined by the subcommands described below.

Remarks:

The DRAW statement combines many of the capabilities of the other graphics statements into the Graphics Macro Language. The Graphics Macro Language defines a set of characteristics that comprehensively describe a particular image. In this case, the characteristics include motion (up, down, left, right), color, angle, and scale factor.

Each of the following subcommands initiates movement from the current graphics position. This is usually the coordinate of the last graphics point plotted with another GML command. The current position defaults to the center of the screen when a program is run.

Prefixes

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

B   Move but don't plot any points.

N   Move but return to original position when done.

Cursor Movement

The following commands specify movement in units. The size of a unit may be modified by the S command. The default unit size is one point. If no argument is supplied, the cursor is moved one unit.

U [<n>] Move up (scale factor *n) points
D [<n>] Move down
L [<n>] Move left
R [<n>] Move right
E [<n>] Move diagonally up and right
F [<n>] Move diagonally down and right
G [<n>] Move diagonally down and left
H [<n>] Move diagonally up and left

Other Commands

M <x,y>

Move absolute or relative. If x is preceded by a plus (+) or minus (–), x and y are added to the current graphics position and connected with the current position by a line. Otherwise, a line is drawn to point x,y from the current cursor position.

A <n>

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

TA <degrees> - rotate <degrees>.

DEGREES must be in the range -360 to 360 degrees. If DEGREES is positive, rotation is counterclockwise. If DEGREES is negative, rotation is clockwise.

Example:

```
FOR D=0 TO 360 'Draw spokes
DRAW "TA=D;NU50
NEXT D
```

C <n>

Set color n. n can range from 0 to 3 in medium resolution graphics mode, and from 0 to 1 in high resolution graphics mode.

S <n>

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

X <string expression>

Execute substring. This powerful command allows you to execute a second substring from a string, much like GOSUB in Microsoft BASIC. You can have one string execute another, which executes a third, and so on.

Numeric arguments can be constants like "123" or "=<variable>" where <variable> is the name of a variable.

P <paintcolor>,<bordercolor>.

<paintcolor> is an integer paint attribute, and <bordercolor> is the integer border attribute. "Tile" painting is not supported in Draw.

Examples:

```
DRAW "U50R50D50L50" 'Draw a box
DRAW "BE10"          'Move up and right into box
DRAW "P1,3"         'Paint interior
```

```
10 U$="U30;"
20 D$="D30;"
30 L$="L40;"
40 R$="R40;"
50 BOX$=U$+R$+D$+L$
60 DRAW "XBOX$;"
```

The statement DRAW "XU$;XR$;XD$;XL$;" would have drawn the same box.

7.36 EDIT COMMAND

Syntax:

EDIT <line number>

Purpose:

To edit the specified line.

Remarks:

When EDIT is used, SLE GW-BASIC types the specified program line and leaves the user in direct mode. The cursor is placed on the first character of the program line.

See Chapter 4, "Writing Programs Using the SLE GW-BASIC Editor," for full details on screen editing capabilities.

7.37 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. SLE GW-BASIC always returns to command level after an END is executed.

Example:

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

7.38 ENVIRON STATEMENT

Syntax:

ENVIRON <string>

Purpose:

To modify a parameter in SLE MS-DOS's Environment String Table.

Remarks:

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

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

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

This statement could be used to change the "PATH" parameter for a child process, or to pass parameters to a child by inventing a new Environment Parameter. (See the SLE MS-DOS Utilities — PATH Command.)

Errors include parameters that are not strings and an "out of memory" when no more space can be allocated to the Environment String Table. The amount of free space in the table will usually be quite small.

Example:

The following SLE MS-DOS command will create a default "PATH" to the root directory on DISK A:

    PATH=A:

The PATH may be changed to a new value by:

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

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

    ENVIRON "SESAME=PLAN"

The Environment String Table now contains:

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

If you then entered:

    ENVIRON "SESAME=;"

then you would have deleted SESAME, and you would have a table containing:

    PATH=A:SALES;A:ACCOUNTING

Also see ENVIRON$ Function and SHELL Command, Sections 7.39 and 7.131, respectively.

7.39 ENVIRON$ FUNCTION

Syntax:

ENVIRON$ (<string parameter>)
ENVIRON$ (<n>)

where n is an integer.

Purpose:

To retrieve a parameter string from BASIC's Environment String Table.

Remarks:

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

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

7.40 EOF FUNCTION

Syntax:

EOF(<file number>)

Purpose:

To test for the end-of-file condition.

Remarks:

Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while inputting, to avoid "Input past end" errors.

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

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

Example:

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

7.41 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 for 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)
.
.
.
```

7.42 ERDEV,ERDEV$ FUNCTIONS

Syntax:

ERDEV ERDEV$

Purpose:

To provide a way to obtain device-specific status information. ERDEV is an integer function which contains the error code returned by the last device to declare an error. ERDEV$ is a string function which contains the name of the Device Driver which generated the error.

Remarks:

These functions may not be set by the programmer.

ERDEV is set by the Interrupt X'24' handler when an error within DOS is detected.

ERDEV will contain the INT 24 error code in the lower eight bits.

Example:

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

```
PRINT ERDEV, ERDEV$
will yield
9        MYLPT2
```

7.43 ERR AND ERL FUNCTIONS

Syntax:

ERR ERL

Remarks:

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

With the SLE GW-BASIC Interpreter, if the statement that caused the error was a direct mode statement, ERL will contain 65535.

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 words, neither may appear to the left of the equal sign in a LET (assignment) statement. SLE GW-BASIC error codes are listed in Appendix B.

Example:

To test whether an error occurred in a direct statement, the user could enter:

```
IF 65535 = ERL THEN PRINT "Direct Error"
```

When testing within a program, use:

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

7.44 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 256. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix B), 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 SLE 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 SLE 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, SLE 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:

```
20 S=15
30 ERROR S
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     (SLE 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
 .
 .
 .
```

7.45 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 or equal to 88.02969, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

The EXP function will return a single precision value unless the / D switch was used when BASIC was invoked and a double precision variable is used as the argument.

Example:

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

will yield

54.59815

7.46 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.

Any field statement which is executed while a file is opened has no effect after that file is closed. For example, assume file 1 is opened, fielded, and closed. If file 1 is re-opened it should be re-fielded.

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 no longer refers to the random file record buffer, but to the variables stored in string space.

Example 1:

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

Allocates the first 20 bytes in the random file buffer to the string variable N$, the next 10 bytes to ID$, and the next 40 to ADD$. FIELD does not place any data in the random file buffer. (See also GET Statement, Section 7.51, and LSET and RSET Statements, Section 7.84.)

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%
30    READ SIZE% (LOOP%)
40 NEXT LOOP%
50 DATA 9,10,11,12,21,41
  .
  .
  .
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%)
```

7.47 FILES STATEMENT

Syntax:

FILES [<filespec>]

where <filespec> includes either a filename or a pathname 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. The asterisk need not be used in the case where all the files on a drive are requested, e.g., FILES "B:".

If a filespec is used, and no explicit path is given, the current directory is the default.

Examples:

FILES

Shows all files on the current directory.

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.

FILES "\SALES"

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

FILES "\SALES\MARY"

Displays MARY <dir> if MARY is a subdirectory of SALES or if MARY is a file, displays its name.

7.48 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

7.49 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, SLE 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 10 STEP 2
30    PRINT I;
40    LET K = K+10
50    PRINT K
60 NEXT I
```

will yield

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

In this example, the loop counter, I, advances +2 on each cycle. The loop prints the counter, increments K, and prints K.

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 I
```

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.

7.50 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 SLE GW-BASIC. Arguments to FRE are dummy arguments.

FRE(" ") forces a garbage collection before returning the number of free bytes.

Remarks:

SLE 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)
might yield
14542
```

7.51 GET STATEMENT — FILE I/O

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 16,777,215.

The GET and PUT statements allow fixed-length input and output for SLE 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.

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. The EOF function may be used after a GET statement to see if that GET was beyond the end of file marker.

7.52 GET STATEMENT — GRAPHICS

Syntax:

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

used with

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

where (x1,y1)-(x2,y2) is a rectangular area on the display screen. The rectangle is defined with (x1,y1) and (x2,y2) being the upper left and the lower right vertices.

<array name> is the name assigned to the place that will hold the image. The array can be any type except string. It must be dimensioned large enough to hold the entire image, otherwise an "Illegal function call" results. Unless the array is type integer, the contents of the array after a GET will be meaningless when interpreted directly (see below).

Purpose:

The GET and PUT statements are used together to transfer graphic images to and from the screen.

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

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

Remarks:

One of the most useful things that can be done with GET and PUT is animation. (See PUT Statement, Section 7.114, for discussion of animation.)

GET reads the colors of the points within the specified rectangle into the array. The specified rectangle has points (*x1,y1*) and (*x2,y2*) as opposite corners. (This is the same as the rectangle drawn by the LINE statement using the B option.)

GET and PUT can be used for high speed object motion in graphics mode. You might think of GET and PUT as "bit pump" operations which move bits onto (PUT) and off of (GET) the screen Remember that PUT and GET are also used for random access files, but the syntax of these statements is different.

The array is used simply as a place to hold the image and must be numeric; it may be any precision, however. The required size of the array, in bytes, is:

4+INT((*x***bitsperpixel*+7)/8)**y*

where *x* and *y* are the lengths of the horizontal and vertical sides of the rectangle, respectively. The value of *bitsperpixel* is 2 in medium resolution and 1 in high resolution.

For example, suppose we want to use the GET statement to get a 10 by 12 image in color mode. The number of bytes required is 4+INT((10*3+ 7)/8) *12, or 52 bytes. The bytes per element of an array are:

- 2 for integer
- 4 for single-precision
- 8 for double-precision

Therefore, we could use an integer array with at least 26 elements.

The information from the screen is stored in the array as follows:

1. two bytes giving the x dimension in bits
2. two bytes giving the y dimension in bits
3. the data itself

It is possible to examine the x and y dimensions and even the data itself if an integer array is used. The x dimension is in element 0 of the array, and the y dimension is in element 1. Keep in mind, however, that integers are stored low byte first, then high byte; but the data is actually transferred high byte first, then low byte.

7.53 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 SLE GW-BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement.

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
```

7.54 GOTO STATEMENT

Syntax:

GOTO <line number>

Purpose:

To branch unconditionally 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
```

7.55 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 (-32768 ~ 65535).

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 7.92, for details on octal conversion.

7.56 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, is 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 X>Y
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" when 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 screen 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 screen.

7.57 INKEY$ FUNCTION

Syntax:

INKEY$

Purpose:

Reads a character from the keyboard containing a character read from the standard input device or a null string if no character is pending there. The keyboard is usually the standard input device.

Remarks:

INKEY$ only reads a single character, even if there are several characters waiting in the keyboard buffer. The returned value is a zero-, one-, or two-character string.

- A null string (length zero) indicates that no character is pending at the keyboard.
- A one-character string contains the actual character read from the keyboard.
- A two-character string indicates a special extended ASCII code. For a complete list of these codes, see Appendix G.

You must assign the result of INKEY$ to a string variable before using the character with any BASIC statement or function.

While INKEY$ is being used, no characters are displayed on the screen and all characters are passed through to the program except for:

- Ctrl-Break/Stop, which stops the program
- Ctrl-Num Lock, which sends the system into a pause state
- Alt-Ctrl-Del, which does a System Reset
- Print, which prints the screen.

Example:

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSES=""
1020 FOR I%=1 TO TIMELIMIT%
1030    A$=INKEY$
1035    IF LEN(A$)=0 THEN 1060
1040    IF ASC(A$)=13 THEN TIMEOUT%=0
1045    IF TIMEOUT%. = 0 THEN RETURN
1050    RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

7.58 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
```

7.59 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.

BASIC can be re-directed to read from standard input and write to standard output by providing the input and output filenames when invoking BASIC. (See Section 2.1, "Invoking BASIC.")

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.

7.60 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 SLE 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
```

7.61 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 standard input device. If input has not been redirected, the keyboard is the standard input device.

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.

BASIC can be re-directed to read from standard input by providing the input filename on the command line. (See Section 2.1, "Invoking BASIC.")

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
.
.
.
```

7.62 INSTR FUNCTION

Syntax:

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

Purpose:

To search for the first occurrence of string Y$ in X$, and to return 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, and if no I was specified, then INSTR returns 1. X$ and Y$ may be string variables, string expressions, or string literals.

Example:

```
10 X$="ABCDE5"
20 Y$="B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
```

will yield

2 6

7.63 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 7.13 and 7.48, respectively, which also return integer values.

7.64 IOCTL STATEMENT

Syntax:

IOCTL [#]<filenumber>, <string>

Purpose:

To transmit a control character or string to a device driver.

Remarks:

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

The IOCTL statement works only if:

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

Most standard MS-DOS device drivers don't process IOCTL strings, and it is necessary for the programmer to determine whether the specific driver can handle the command.

Example:

If a user wanted to set the page length to 66 lines per page on LPT1, the procedure might be:

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

Also see the IOCTL$ Function.

7.65 IOCTL$ FUNCTION

Syntax:

IOCTL$ ([#]<filenumber>)

Purpose:

To receive a control data string from a device driver.

Remarks:

The IOCTL$ function is most frequently used to receive acknowledgement that an IOCTL statement succeeded or failed, or to obtain current status information.

IOCTL$ could be used to ask a communications device to return the current baud rate, information on the last error, logical line width, etc.

The IOCTL$ function works only if:

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

Example:

```
10 OPEN "  DEV  F00" AS #1
20 IOCTL #1, "RAW"
```

This example tells the device that the data is raw.

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

In this continuation, if the Character Driver 'F00' responds 'false' from the raw data mode IOCTL statement, then the file is closed.

Also see the IOCTL Statement.

7.66 KEY STATEMENT

Syntax:

```
KEY n, X$
KEY LIST
KEY ON
KEY OFF
```

n is the number of the function key (in the range 1 to 12).

X$ is the text assigned to the specified key.

Purpose:

To assign softkey values to function keys and display the values.

Remarks:

The KEY statement allows function keys to be designated for special "softkey" functions. Each of the function keys may be assigned a 15-byte string which will be input to SLE GW-BASIC when that key is pressed.

Initially, the "softkeys" are assigned the following values:

F1	LIST	F2	AUTO
F3	RUN←	F4	LOAD"
F5	SAVE"	F6	CONT←
F7	,"LPT1:"	F8	TRON←
F9	TROFF←	F10	KEY
F11	EDIT .←	F12	SCREEN 0,0,0←

The arrow (←) indicates Enter and ( ) indicates space.

Softkeys can be displayed with the KEY ON, KEY OFF, and KEY LIST statements.

KEY ON causes the softkey values to be displayed on the bottom line of the screen.

KEY OFF erases the softkey display from the bottom line, making that line available for program use. It does not disable the function keys.

KEY LIST displays all softkey values on the screen, with all 15 characters of each key displayed.

Assigning a null string (string of length 0) to a softkey disables the function key as a softkey.

If the function key number is not in the range of permissible function key numbers, an "Illegal function call" error is produced, and the previous key string expression is retained.

When a softkey is assigned, the INKEY$ function returns one character of the softkey string per invocation. If the softkey is disabled, INKEY$ returns a two-character string.

Example:

```
50 KEY ON 'Displays the softkey on bottom line
60 KEY OFF ' Erases softkey display
70 KEY 1,"MENU"+CHR$(13) '
```

Assigns the string "MENU" followed by a carriage return to softkey 1.

Such assignments might be used to speed data entry.

```
80 KEY 1,""   'Disables softkey 1
```

The following routine initializes the first five softkeys:

```
10 KEY OFF 'Turns off key display during initialization
20 DATA "EDIT ","LET ","SYSTEM","PRINT ","LPRINT "
30 FOR I = 1 TO 5
40   READ SOFTKEYS$(I)
50   KEY I,SOFTKEY$(I)
60 NEXT I
70 KEY ON 'Displays new softkeys
```

7.67 KEY(n) STATEMENT

Syntax:

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

where (n) is the number of a function key, help key, a user-defined key. These keys are numbered sequentially after the function keys (1~10) in the following order: up (11), left (12), right (13), down (14); then come the user-defined keys (15~20).

User Defined keys are defined by the statement:

KEY i,CHR$(j)+CHR$(k)

i is the number of a user-defined key in the range 15 to 20.

j is a scan code in the range 1 to 83. See Appendix E, "Keyboard Diagram and Scan Codes."

k is an ASCII character code in the range 1 to 255. See Appendix A, "ASCII Character Code."

Purpose:

To enable or disable event trapping of softkey or cursor direction key activity for the specified trappable key.

Remarks:

Note that the KEY statement described in Section 7.66 assigns softkey 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.

KEY(n) ON enables softkey or cursor direction key event trapping by an ON KEY statement (see ON KEY Statement, Section 7.96). While trapping is enabled, and if a non-zero line number is specified in the ON KEY statement, SLE GW-BASIC checks between every statement to see if a softkey or cursor direction key has been used. If it has, the ON KEY statement is executed. The text that would normally be associated with a function key will not be printed.

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

Pressing Ctrl-Break/Stop disables all active key traps. For additional information on key event trapping, see "Event Trapping," Section 6.2, and "ON KEY Statement," Section 7.96.

Example:

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

7.68 KILL STATEMENT

Syntax:

KILL [<filespec>]

Purpose:

To delete a file or a pathname 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.

Since it is possible to reference the same file in a sub-directory via different paths, it is nearly impossible for BASIC to know that it is indeed the same file simply by looking at the path. For example, if MARY is your current directory, then:

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

all refer to the same file. Therefore, any open file with the same file name will cause a "file already open" error.

WARNING

Be extremely careful when using wildcards with this command.

Examples:

200 KILL "DATA1?.DAT"

The position taken by the question mark will match any valid filename character. This command will kill any file that has a six character name starting with "DATA1" and has the filename extension ".DAT". This includes "DATA10.DAT" and "DATA1Z.DAT".

210 KILL "DATA1.*"

Kills all files named DATA1, regardless of the filename extension.

220 KILL "..\GREG*.DAT"

Kills all files with the extension ".DAT" in a directory called GREG.

7.69 LEFT$ FUNCTION

Syntax:

LEFT$(<string>,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 <string>, (LEN(X$)), the entire string (<string>) will be returned. If I = 0, the null string (length zero) is returned.

Example:

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

will yield

BASIC

Also see the MID$ and RIGHT$ functions, Sections 7.87 and 7.123, respectively.

7.70 LEN FUNCTION

Syntax:

LEN(<string>)

Purpose:

To return the number of characters in <string>. Nonprinting characters and blanks are counted.

Example:

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

will yield

16

7.71 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
 .
 .
 .
```

7.72 LINE STATEMENT

Syntax:

LINE [[STEP](x1,y1)]-[STEP](x2,y2)
[,[<color>][,b[f]]][,style]

(x1,y1) is the coordinate for the starting point of the line.

(x2,y2) is the ending point for the line.

The [STEP] option makes the specified coordinates relative to the "most recent point," instead of absolute, mapped coordinates.

<color> is the color number in the range 0 to 7 and selects the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the current foreground color.

,b draws a box with the points (x1,y1) and (x2,y2) specifying the upper left and lower right corners.

,bf draws a filled box.

,style is a 16-bit integer mask used when putting pixels down on the screen. This is called "line styling."

Purpose:

To draw a line or box on the screen.

Remarks:

When coordinates specify a point that is not in the current viewport, the line segment is clipped to the viewport.

The relative coordinate form STEP (xoffset,yoffset) can be used in place of an absolute coordinate. For example, assume that the most recent point referenced was (10,10). The statement LINE STEP (10,5) would specify a point at offset 10 from x and offset 5 from y, that is, (20,15).

If the STEP option is used for the second coordinate on a LINE statement, it is relative to the first coordinate in the statement. Other ways to establish a new "most recent point" are to initialize the screen with the CLS and SCREEN statements (Sections 7.17 and 7.128, respectively). Using the PSET, PRESET, CIRCLE and DRAW statements will establish a new "most recent point."

Each time LINE stores a point on the screen, it uses the current circulating bit in [style]. If that bit is a 0, then no storing will be done; if the bit is a 1, the point is stored. After each point is stored, the next bit position in [style] is selected. Since a 0 bit in [style] causes no change to the point on the screen, the user may prefer to draw a background line before a 'styled' line in order to force a known background. Style is used for normal lines and boxes, but has no effect on filled boxes.

Examples:

The following examples assume a screen 320 pixels wide by 200 pixels high.

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

Draws a line from the last point to x2,y2 in the foreground color.

```
20 LINE (0,0)-(319,199)
```

Draws a diagonal line across the screen (downward).

```
30 LINE (0,100)-(319,100)
```

Draws a line across the screen.

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

Draws a line in color 2.

```
10 FOR x=0 to 319
20 LINE (x,0)-(x,199),x AND 1
```

Draws an alternating line on-line off pattern on a monochrome display.

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

Draws a box in the foreground (note that the color is not included).

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

Draws a filled box in color 2. Coordinates are given as offsets.

```
10 LINE (0,0)-(160,100),3,,&HFF00
```

Draws a dashed line from the upper left hand corner to the center of the screen.

7.73 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-Break/Stop. SLE 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 7.74.

7.74 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 an SLE GW-BASIC program saved in ASCII format is being read as data by another program. (See SAVE Command, Section 7.127.)

When SLE GW-BASIC is invoked with redirected input and output, all LINE INPUT statements will read from the input file specified instead of the keyboard.

When input is redirected, SLE GW-BASIC will continue to read from this source until a control-Z is detected. This condition may be tested with the EOF function. If the file is not terminated by a control-Z, or a BASIC file input statement tries to read past end-of-file, then any open files are closed, the message "Read past end" is written to standard output, and BASIC returns to SLE MS-DOS.

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
```

7.75 LIST COMMAND

Syntax:

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

<line number> is in the range 0 to 65529.

<device> is a device designation string, such as SCRN: or LPT:, or a filename.

Purpose:

To list all or part of the program currently in memory.

Remarks:

SLE GW-BASIC always returns to command level after a LIST is executed.

If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either when the end of the program is reached or by typing Break.) If <line number> is included, only the specified line will be listed.

If only the first <line number> is specified, that line and all higher-numbered lines are listed.

If only the second <line number> is specified, all lines from the beginning of the program through that line are listed.

If both <line number(s)> are specified, the entire range is listed.

If the <device> is omitted, the listing is shown at the terminal.

Examples:

```
LIST
```

Lists the program currently in memory.

```
LIST 500
```

Lists line 500.

```
LIST 150-
```

Lists all lines from 150 to the end.

```
LIST -1000
```

Lists all lines from the lowest number through 1000.

```
LIST 150-1000
```

Lists lines 150 through 1000, inclusive.

```
LIST 150-1000,"LPT:"
```

Lists lines 150 through 1000 on the line printer.

7.76 LLIST COMMAND

Syntax:

LLIST [<line number>[-[<line number>]]]

Purpose:

To list all or part of the program currently in memory on the line printer.

Remarks:

LLIST assumes a 132-character-wide printer.

SLE GW-BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for the LIST Command, Section 7.75.

Example:

See the examples for the LIST Command, Section 7.75. With the exception of the last one, which addresses a device, LLIST will work in a similar way.

7.77 LOAD COMMAND

Syntax:

LOAD <filespec>[,R]

Purpose:

To load a file from an input device into memory.

Remarks:

For loading a program, the <filespec> is an optional device specification followed by a filename or pathname that conforms to SLE MS-DOS'S rules for filenames. BASIC appends the default filename extension .BAS if the user specifies no extensions, when the file is saved to the disk.

The <filespec> must include the filename that was used when the file was saved, or created by an editor. (BASIC will 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.BAS

LOAD "B:MYPROG"

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

7.78 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 actual record number within the file.

With sequential files, LOC returns the current byte position in the file, divided by 128.

Remarks:

When a file is opened for APPEND or OUTPUT, LOC returns the size of the file in (bytes/128).

For a communications file, LOC(X) 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
```

7.79 LOCATE STATEMENT

Syntax:

LOCATE [row][,[col][,[cursor][,[start][,stop]]]]

<row> is a numeric expression in the range 1 to 25. It indicates the screen line number where you want to place the cursor.

<col> is a numeric expression in the range 1 to 40 or 1 to 80, depending upon screen width. It indicates the screen column number where you want to place the cursor.

<cursor> is a value indicating whether the cursor is visible or not. A 0 (zero) indicates off, 1 (one) indicates on.

<start> is the cursor starting scan line. It must be a numeric expression in the range 0 to 31.

<stop> is the cursor stop scan line. It also must be numeric expression in the range 0 to 31.

cursor, *start* and *stop* do not apply to graphics mode.

Purpose:

Positions the cursor on the active screen. Optional parameters turn the blinking cursor on and off and define the size of the blinking cursor.

Remarks:

start and *stop* allow you to make the cursor any size you want. You indicate the starting and ending scan lines. The scan lines are numbered from 0 at the top of the character position. The bottom scan line is 7. If *start* is given and *stop* is omitted, *stop* assumes the value of *start*. If *start* is greater than *stop*, you'll get a two-part cursor. The cursor "wraps" from the bottom line back to the top.

After a LOCATE statement, I/O statements to the screen begin placing characters at the specified location.

When a program is running, the cursor is normally off. You can use LOCATE ,,1 to turn it back on.

Normally, BASIC will not print to line 25. However, you can turn off the soft key display using KEY OFF, then using LOCATE 25,1: PRINT... to put things on line 25.

Any parameters may be omitted. Omitted parameters assume the current value.

Any values entered outside of the ranges indicated will result in an "Illegal function call" error. Previous values are retained.

Example:

```
10 LOCATE 1,1
```

Moves the cursor to the home position in the upper left-hand corner of the screen.

```
20 LOCATE ,,1
```

Makes the blinking cursor visible; its position remains unchanged.

```
30 LOCATE ,,,7
```

Position and cursor visibility remain unchanged. Sets the cursor to display at the bottom of the character starting and ending on scan line 7.

```
40 LOCATE 5,1,1,0,7
```

Moves the cursor to line 5, column 1. Makes the cursor visible, covering the entire character cell on the Color/Graphics Monitor Adapter, starting at scan line 0 and ending on scan line 7.

7.80 LOF FUNCTION

Syntax:

LOF(<file number>)

Purpose:

To return the length of the named file in bytes.

Remarks:

When a file is opened for APPEND or OUTPUT, LOF returns the size 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.

7.81 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
```

7.82 LPOS FUNCTION

Syntax:

LPOS(X)

where X is the index of the printer being tested; that is LPT1: would be tested with LPOS(1), LPT2: with LPOS(2), etc.

Purpose:

To return the current position of the printer's print head within the 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)
```

7.83 LPRINT AND LPRINT USING STATEMENTS

Syntax:

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

Purpose:

To print data on the printer.

Remarks:

Same as PRINT and PRINT USING, except output goes to the line printer, and the file number option is not permitted. See Sections 7.109 and 7.110, respectively.

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

Examples:

See Sections 7.109 and 7.110.

7.84 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) or to left- or right-justify the value of a string into a string variable.

Remarks:

<string variable> is the name of a variable defined in a FIELD statement.

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$ functions, Section 7.89.

Examples:

```
150 LSET A$=MKS$(AMT)
160 LSET D$=MKI$(COUNT%)
```

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.

7.85 MERGE COMMAND

Syntax:

MERGE <filespec>

Purpose:

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

Remarks:

For merging a program not in memory, the <filespec> is an optional device specification followed by a filename or pathname that conforms to SLE MS-DOS's rules for filenames. BASIC appends the default filename extension ".BAS" if the user specifies no extensions, and the file has been saved to the disk.

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.)

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

Example:

MERGE "NUMBRS"

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

7.86 MID$ STATEMENT

Syntax:

MID$(<string 1>,n[,m])=<string 2>

where n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

Purpose:

To replace a portion of one string with another string.

Remarks:

The characters in <string 1>, beginning at position n, are replaced by the characters in <string 2>. The optional "m" refers to the number of characters from <string 2> that will be used in the replacement. If "m" is omitted, all of <string 2> is used. However, regardless of whether "m" is omitted or included, the replacement of characters never goes beyond the original length of <string 1>.

Example:

```
10 A$=KANSAS CITY, MO"
20 MID$(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 7.87.

7.87 MID$ FUNCTION

Syntax:

MID$(<string>,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 <string>, that is, (LEN(<string>)), 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 7.69 and 7.123, respectively.

7.88 MKDIR STATEMENT

Syntax:

MKDIR <pathname>

Purpose:

To create a new directory.

Remarks:

<pathname> is a string expression specifying the name of the directory which is to be created. MKDIR works exactly like the SLE MS-DOS command MKDIR. The <pathname> must be a string of less than 128 characters. (See Section 5.5, "File Handling," for a discussion of tree-structured directories.)

Example:

Assume the current directory is the root.

MKDIR "SALES"

Creates a sub-directory named SALES in the current directory of the current drive.

MKDIR "B:USERS

Creates a sub-directory named USERS in the current directory of drive B.

Also see the CHDIR and RMDIR statements, Sections 7.11 and 7.124, respectively.

7.89 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 7.25.

7.90 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. Also, both files must be on the same drive.

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 with the new name.

NAME may not be used to rename directories.

<old filename> must be closed before the renaming command is executed. Also, there must be one free file handle.

Examples:

NAME "ACCTS" AS "LEDGER"

In this example, the file that was formerly named ACCTS will now be named LEDGER.

NAME may be used to move a file from one directory to another. For example:

NAME "\X\CLIENTS" AS "\XYZ\P\CLIENTS"

7.91 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. SLE GW-BASIC always returns to command level after a NEW is executed.

NEW closes all files and turns tracing off.

Example:

NEW

7.92 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 (-32768~65535).

Example:

```
PRINT OCT$(24)
will yield
30
```

See the HEX$ function, Section 7.55, for details on hexadecimal conversion.

7.93 ON COM STATEMENT

Syntax:

ON COM(1) GOSUB <line number>

where <line number> is the number of the first line of a subroutine that is to be performed when activity occurs on the communications port.

Purpose:

To specify the first line number of a subroutine to be performed when activity occurs on a communications port.

Remarks:

A <line number> of zero disables the communications event trap.

The ON COM statement will only be executed if a COM(1) ON statement has been executed (see COM Statement, Section 7.19) to enable event trapping. If event trapping is enabled, and if the <line number> in the ON COM statement is not zero, SLE GW-BASIC checks between statements to see if communications activity has occurred on the specified port. If communications activity has occurred, a GOSUB will be performed to the specified line.

If a COM OFF statement has been executed for the communications port (see COM Statement, Section 7.19), the GOSUB is not performed and is not remembered.

If a COM STOP statement has been executed for the communications port (see COM Statement, Section 7.19), the GOSUB is not performed, but will be performed as soon as a COM ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic COM STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a COM ON statement unless an explicit COM OFF was performed inside the subroutine.

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 SLE GW-BASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs, or when Control-Break/Stop is used.

7.94 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 SLE 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
```

7.95 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 number 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 either zero or greater than the number of items in the list, control drops to the next BASIC 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
```

7.96 ON KEY STATEMENT

Syntax:

ON KEY(n) GOSUB <line number>

(n) is the number of a function key, direction key, or user-defined key.

<line number> is the number of the first line of a subroutine that is to be performed when the specified function or cursor direction key is pressed.

Purpose:

To specify the first line number of a subroutine to be performed when a specified key is pressed.

Remarks:

A <line number> of zero disables the event trap.

The ON KEY statement will only be executed if a KEY(n) ON statement has been executed (see KEY(n) Statement, Section 7.67) to enable event trapping. If key trapping is enabled, and if the <line number> in the ON KEY statement is not zero, SLE GW-BASIC checks between statements to see if the specified function, user-defined or cursor direction key has been pressed. If so, the program will branch to a subroutine specified by the GOSUB statement.

If a KEY(n) OFF statement has been executed for the specified key (see KEY(n) Statement, Section 7.67), the GOSUB is not performed and is not remembered.

If a KEY STOP statement has been executed for the specified key (see KEY(n) Statement, Section 7.67), the GOSUB is not performed, but will be performed as soon as a KEY(n) ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic KEY(n) STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a KEY(n) ON statement unless an explicit KEY(n) OFF was performed inside the subroutine.

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" could result.

Event trapping does not take place when SLE GW-BASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs, or when Control-Break/Stop is used.

The following rules apply to keys trapped by SLE GW-BASIC:

1. The line printer echo toggle key is processed first. Defining this key as a user defined key trap will not prevent characters from being echoed to the line printer if depressed.
2. Function keys and the cursor direction keys are examined next. Defining a function key or cursor direction key as a user defined key trap will have no effect as they are considered pre-defined.
3. Finally, the user defined keys are examined.
4. Any key that is trapped is not passed on. That is, the key is not read by SLE GW-BASIC.

NOTE

When a key is trapped, that occurrence of the key is destroyed. Therefore, you cannot subsequently use the INPUT or INKEY$ statements to find out which key caused the trap. So if you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

The ON KEY(n) statement allows 6 additional user defined KEY traps. This allows any key, Control-key, Shift-key, or Alt-key to be trapped by the user as follows:

ON KEY(i) GOSUB <line_number>

Where: <i> is an integer expressing a legal user-defined key number.

Example:

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

In the above, the programmer has overriden the normal function associated with function key 4, and replaced it with "SCREEN 0,0", which will be printed whenever that key is pressed. The value may be reassigned and it will resume its standard function when the machine is rebooted.

```
100 KEY 15, CHR$(&H04) + CHR$(83)
110 KEY 15 ON
        .
        .
        .
1000 PRINT "If you wnat to stop processing for a break"
1010 PRINT "press the Control key and Break/Stop at the"
1020 PRINT "same time"
1030 ON KEY(15) GOSUB 3000.
        .
   Operator presses Control-Break/Stop
        .
3000 REM ** Suspend processing loop.
3010 CLOSE #1
3020 RESET
3030 CLS
3035 PRINT "Enter CONT to continue."
3040 STOP
3050 OPEN "A", #1, "ACCOUNTS.DAT"
3060 RETURN
```

In the above, the programmer has enabled Control-Break/Stop to enter a subroutine which closes the files and stops program execution until the operator is ready to continue.

7.97 ON TIMER STATEMENT

Syntax:

ON TIMER (n) GOSUB <line number>

Purpose:

To provide an event trap during real time.

Remarks:

ON TIMER causes an event trap every (n) seconds. (n) must be a numeric expression in the range of 1 to 86400 (1 second to 24 hours). Values outside this range generate an "Illegal function call" error.

The ON TIMER statement will only be executed if a TIMER ON statement has been executed to enable event trapping. If event trapping is enabled, and if the <line number> in the ON TIMER statement is not zero, SLE GW-BASIC checks between statements to see if the time has been reached. If it has, a GOSUB will be performed to the specified line.

If a TIMER OFF statement has been executed, the GOSUB is not performed and is not remembered.

If a TIMER STOP statement has been executed, the GOSUB is not performed, but will be performed as soon as a TIMER ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic TIMER STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a TIMER ON statement unless an explicit TIMER OFF was performed inside the subroutine.

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.

Example:

Display the time of day on line 1 every minute.

```
10 ON TIMER(60) GOSUB 10000
20 TIMER ON
        .
        .
        .
10000 LET OLDROW=CSRLIN 'Save current Row
10010 LET OLDCOL=POS(0) 'Save current Column
10020 LOCATE 1,1:PRINT TIME$;
10030 LOCATE OLDROW,OLDCOL 'Restore Row & Col
10040 RETURN
```

Also see TIMER ON, TIMER OFF and TIMER STOP Statements, Section 7.146.

7.98 OPEN STATEMENT

Syntaxes:

OPEN <mode1>,[#]<file number>, <filespec>
[,<record length>]

OPEN <filespec> (FOR <mode2>) AS [#]<file number>
[LEN=<record length>]

<filespec> is an optional device specification followed by a filename or pathname, that conforms to SLE MS-DOS's rules for filenames.

<device> is a character device.

<mode1> is a string expression. The first character must be 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.

<mode2> is an expression which is one of the following:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.
APPEND	Specifies sequential output mode and sets the file pointer at the end of file and the record number as the last record of the file. A PRINT# or WRITE# statement will then extend (append) the file.

If <mode2> is omitted, the default random access mode is assumed. Random, however, cannot be expressed explicitly as the file mode.

<file number> is an integer expression whose value is between 1 and 255. 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.

<record length> is an integer expression that, if included, sets the record length for random files. SLE GW-BASIC will ignore this option if it is used in a statement to OPEN a sequential file. The default length for records is 128 bytes, unless the command line options /I and /R have been used (see Section 2.2, "Command Line Option Switches").

Purpose:

To allow I/O to a file or device.

Remarks:

Files

A file must be opened before any 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.

OPEN allows <pathname> in place of <filespec>. If the pathname is used, and a drive is specified, the drive must be specified at the beginning of the pathname. That is, "B:\SALES\JOHN" is legal, while "\SALES\B: JOHN" is NOT legal.

The LEN= option is ignored if the file being opened has been specified as a sequential file.

Since it is possible to reference the same file in a sub-directory via different paths, it is nearly impossible for BASIC to know that it is the same file simply by looking at the path. For this reason, BASIC will not let you open the file for OUTPUT or APPEND if it is on the same disk even if the path is different. For example, if MARY is your current directory, then:

```
OPEN   "REPORT"
OPEN   "\SALES\MARY\REPORT"...
OPEN   "..\MARY\REPORT" ...
OPEN   "..\..\MARY\REPORT" ...
```

all refer to the same file.

MS-DOS Devices

BASIC devices are:

KYBD:	LPT1:
SCRN:	
COM1:	

The BASIC file I/O system allows the user to take advantage of user installed devices (see the SLE MS-DOS manual for information on character devices).

Character devices opened are opened and used in the same manner as disk files. However, characters are not buffered by BASIC as they are for disk files. The record length is set to one.

BASIC only sends a CR (carriage return X'0D') as end of line. If the device requires a LF (line feed X'0A'), the driver must provide it. When writing device drivers, keep in mind that BASIC users will want to read and write control information. Writing and reading of device control data is handled by the BASIC IOCTL statement and IOCTL$(f) function.

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
```

If a user writes and installs a device called FOO, then the OPEN statement might appear as:

```
10 OPEN "\DEV\FOO" FOR OUTPUT AS #1
```

To open the printer for output, the user could use the line:

```
100 OPEN "LPT:" FOR OUTPUT AS #1
```

which uses the SLE GW-BASIC device driver, or as part of a pathname as in:

```
100 OPEN "\DEV\LPT1" FOR OUTPUT AS #1
```

which uses the SLE MS-DOS device driver.

7.99 OPEN COM STATEMENT

Syntax:

OPEN "COMn: [<speed>][,[<parity>]
[,[<data>][,[<stop>][,RS][,CS[n]][,DS[n]]
[,CD[n]] [,BIN] [,ASC][,LF]]]]" [FOR <mode> AS
[#]<file number> [LEN= <record length>]

COMn: is the name of the device to be opened.

n is the number of a legal communications device, i.e., COM1:.

<speed> is the baud rate, in bits per second, of the device to be opened (75, 150, 300, 600, 1200, 2400, 4800 and 9600).

<parity> designates the parity of the device to be opened. Valid entries are: N (none), E (even), O (odd).

<data> designates the number of data bits per byte. Valid entries are: 5, 6, 7, or 8.

<stop> designates the stop bit. Valid entries are: 1, or 2.

RS suppresses RTS (Request To Send).

CS[n] controls CTS (Clear To Send).

DS[n] controls DSR (Data Set Ready).

CD[n] controls CD (Carrier Detect).

LF specifies that a linefeed is to be sent after a carriage return. See "Remarks" for further discussion of LF.

BIN opens the device in binary mode. BIN is selected by default unless ASC is specified. See "Remarks" for further discussion of BIN.

ASC opens the device in ASCII mode. See "Remarks" for further discussion of ASC.

<mode> is one of the following string expressions:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.

If the <mode> expression is omitted, it is assumed to be random input/output. Random cannot, however, be explicitly chosen as <mode>.

<file number> is the number of the file to be opened.

Purpose:

To open and initialize a communications channel for input/output.

Remarks:

The OPEN COM statement must be executed before a device can be used for RS232 communication.

Any syntax errors in the OPEN COM statement will result in a "Bad File name" error.

The <speed>, <parity>, <data>, and <stop> options must be listed in the order shown in the above syntax. The remaining options may be listed in any order, but they must be listed after the <speed>, <parity>, <data>, and <stop> options.

A "Device timeout" error will occur if Data Set Ready (DSR) is not detected.

LF allows communication files to be printed on a serial line printer. When LF is specified, a linefeed character (OAH) is automatically sent after each carriage return character (OCH). This includes the carriage return sent as a result of the width setting. Note that INPUT# and LINE INPUT#, when used to read from a COM file that was opened with the LF option, stop when they see a carriage return, ignoring the linefeed.

The LF option is superseded by the BIN option.

In the BIN mode, tabs are not expanded to spaces, a carriage return is not forced at the end-of-line, and Control-Z is not treated as end-of-file. When the channel is closed, Control-Z will not be sent over the RS232 line. The BIN option supersedes the LF option.

In ASC mode, tabs are expanded, carriage returns are forced at the end-of-line, Control-Z is treated as end-of-file, and XON/XOFF protocol (if supported) is enabled. When the channel is closed, Control-Z will be sent over the RS232 line.

Example:

```
10 OPEN "COM1:9600,N,8,1,GIN" AS #2
```

will open communications channel 1 in random mode at a speed of 9600 baud with no parity bit, 8 data bits, and 1 stop bit. Input/Output will be in the binary mode. Other lines in the program may now access channel 1 as file number 2.

7.100 OPTION 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 any arrays.

Chained programs may have an OPTION BASE statement if no arrays are passed between them or the specified base is identical in the chained programs. The chained program will inherit the OPTION BASE value of the chaining program.

Example:

```
10 OPTION BASE 1
```

7.101 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
```

7.102 PAINT STATEMENT

Syntax:

PAINT (<x>,<y>)[,<paint attribute>
[,<border color>] [,background attribute]]

(<xstart> and <ystart>) are the coordinates where painting is to begin. Painting should always start on a non-border point. If painting starts within a border, the bordered figure is painted. If painting starts outside a bordered figure, the background is painted.

If the <paint attribute> is a string expression PAINT will execute "Tiling," a process similar to "Line-styling." Like LINE, PAINT looks at a "tiling" mask each time a point is put down on the screen.

If <paint attribute> is a numeric expression, then the number must be a valid color and is used to paint the area as before (see COLOR Statement, Section 7.18). If the <paint attribute> is not specified, the foreground color will be used.

<border color> identifies the border color of the figure to be filled. When the order color is encountered, painting of the current line will stop. If the <border color> is not specified, the <paint attribute> will be used.

<background attribute> is a string formula returning character data. When it is omitted, the default is CHR$(0).

When specified, <background attribute> gives the "background tile slice" to skip when checking for termination of the boundary. Painting is terminated when adjacent points display the paint color; specifying a background tile slice allows the user to paint over an already painted area without terminating the process because two consecutive lines with the same paint attributes are encountered.

Purpose:

To fill a graphics area with the color or pattern specified.

Remarks:

Painting is complete when a line is painted without changing the color of any pixel; i.e., the entire line is equal to the paint color.

The PAINT command can be used to fill any figure, but painting complex figures may result in an "Out of Memory" error. If this happens, the CLEAR statement may be given to increase the amount of stack space available.

The PAINT command permits coordinates outside the screen or viewport.

Tiling

Tiling is the design of a PAINT pattern that is 3 bits wide and up to 64 bytes long. Each byte in the Tile String masks 8 bits along the x axis when putting down points. Construction of this Tile mask works as follows:

Use the syntax PAINT (x,y), CHR$(n)...CHR$(n) where (n) is a number between 0 and 255 which will be represented in binary across the x-axis of the "tile". Each CHR$(n) up to 64 will generate an image not of the assigned character, but of the bit arrangement of the code for that character. For example, the decimal number 85 is binary "01010101"; the graphic image line on a black and white screen generated by CHR$(85) is an eight pixel line, with even numbered points turned white, and odd ones black. That is, each bit containing a "1" will set the associated pixel on and each bit filled with a "0" will set the associated bit off in a black and white system. The ASCII character CHR$(85), which is "U," is not displayed in this case.

If the current screen mode supports only two colors, then the screen can be painted with 'X's with the following statement.

```
PAINT (320,100),CHR$(129)+CHR$(66)
+CHR$(36)+CHR$(24)+ CHR$(24)+CHR$(36)+
CHR$(66)+CHR$(129)
```

This appears on the screen as:

```
                x increases
0,0   x                          x    CHR$(129)   Tile byte 1
0,1      x                    x       CHR$(66)    Tile byte 2
0,2         x              x          CHR$(36)    Tile byte 3
0,3            x        x             CHR$(24)    Tile byte 4
0,4            x        x             CHR$(24)    Tile byte 5
0,5         x              x          CHR$(36)    Tile byte 6
0,6      x                    x       CHR$(66)    Tile byte 7
0,7   x                          x    CHR$(129)   Tile byte 8
```

In four-color graphics mode which is selected palette-0 or palette-1 in the COLOR statement, one tile byte describes four pixels. Entry two bits of the tile byte describes one of four possible color associated with each of the four pixels to be plotted.

The following chart shows the binary and hexadecimal values associated with the given colors.

COLOR PALETTE 0	COLOR PALETTE 1	COLOR NUMBER IN BINARY	PATTERN TO DRAW SOLID LINE IN BINARY	PATTERN TO DRAW SOLID LINE IN HEXADECIMAL
green	cyan	01	01010101	&H55
red	magenta	10	10101010	&HAA
yellow	white	11	11111111	&HFF

The following example plots a pattern of boxes with a border color of red in palette 0 and magenta in palette 1.

```
PAINT (320,100),CHRS(&HAA)+CHRS(&H82)+CHRS(&H82)
+CHRS(&H82)+CHRS(&H82)+CHRS(&H82)+CHRS(&H8s)
+CHRS(&HAA)
```

	7	6	5	4	3	2	1	0	
Tile byte 0	1	0	1	0	1	0	1	0	CHRS(&HAA)
Tile byte 1	1	0	0	0	0	0	1	0	CHRS(&H82)
Tile byte 2	1	0	0	0	0	0	1	0	CHRS(&H82)
Tile byte 3	1	0	0	0	0	0	1	0	CHRS(&H82)
Tile byte 4	1	0	0	0	0	0	1	0	CHRS(&H82)
Tile byte 5	1	0	0	0	0	0	1	0	CHRS(&H82)
Tile byte 6	1	0	0	0	0	0	1	0	CHRS(&H82)
Tile byte 7	1	0	1	0	1	0	1	0	CHRS(&HAA)

7.103 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 7.31). For the interpretation of a negative value of I, see VARPTR Function, Section 7.150.

PEEK is the complementary function of the POKE statement.

Example:

```
A=PEEK(&H5A00)
```

In this example, the value at the location with the hex address 5A00 is loaded into a variable A.

7.104 PMAP FUNCTION

Syntax:

PMAP <expression>, <function>

Purpose:

To map world coordinate expressions to physical locations or to map physical expressions to a world coordinate location.

<function> =

0 Maps world expression to physical x coordinate.

1 Maps world expression to physical y coordinate.

2 Maps physical expression to world x coordinate.

3 Maps physical expression to world y coordinate.

Remarks:

The four PMAP functions allow the user to find equivalent point locations between the world coordinates created with the WINDOW statement and the physical coordinate system of the screen or viewport as defined by the VIEW statement.

Examples:

If the statement

SCREEN 1: WINDOW (-1,-1)-(1,1)

is in effect, we can use PMAP to map the world coordinate points of (-1,-1) and (1,1) to their corresponding physical points on the screen.

PMAP(-1,0) returns the physical x coordinate value of 0.

PMAP(-1,1) returns the physical y coordinate value of 199.

PMAP(1,0) returns the physical x coordinate value of 319.

PMAP(1,1) returns the physical y coordinate value of 0.

The above information tells us that the point (-1,-1) which is in the lower left corner of the screen corresponds to the physical point (0,199). We also know that the point (1,1) which is in the upper right corner corresponds to the physical point (319,0).

With the statement WINDOW(-1,-1)-(1,1) the center point of the window is (0,0). PMAP(1,0) returns a physical x coordinate of 160. PMAP(0,1) returns a physical y coordinate of 100. So the point (0,0) which is in the center corresponds to the physical point (160,100).

7.105 POINT FUNCTION

Syntax:

POINT (<xcoordinate>,<ycoordinate>)

<xcoordinate> and <ycoordinate> are the coordinates of the pixel that is to be referenced.

Or

POINT (<function>)

Purpose:

POINT (x,y) allows the user to read the color number of a pixel from the screen. If the specified point is out of range, the value -1 is returned.

POINT with one argument allows the user to retrieve the current Graphics cursor coordinates. Therefore:

x= POINT(FUNCTION)

Returns the value of the current x or y. Graphics accumulator as follows:

function =

0 Returns the current physical x coordinate.

1 Returns the current physical y coordinate.

2 Returns the current logical x coordinate. If the WINDOW statement has not been used, this will return the same value as the POINT(0) function.

3 Returns the current logical y coordinate if WINDOW is active, else returns the current physical y coordinate as in 1 above.

where the physical coordinate is the coordinate on the screen or current viewport.

Examples:

```
10 SCREEN 1
20 LET C=3
30 PSET (10,10),C
40 IF POINT(10,10)=C THEN PRINT "This point is color ";C
```

```
 5 SCREEN 2
10 IF POINT(i,i)<>0 THEN PRESET (i,i) ELSE PSET (i,i)
'invert current state of a point
20 PSET (i,i),1-POINT(i,i) 'another way to invert a point if the system
is black and white.
```

7.106 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. J must be 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 set by the last DEF SEG statement (see Section 7.31). For interpretation of negative values of I, see VARPTR Function, Section 7.150.

The complementary function to POKE is PEEK. (See Section 7.103.)

WARNING

Use POKE carefully. If it is used incorrectly, it can cause SLE GW-BASIC or SLE MS-DOS to crash.

Example:

```
10 POKE &H5A00,&HFF
```

7.107 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 vertical line position of the cursor, use the CSRLIN function (Section 7.24).

Example:

```
IF POS(X)>60 THEN PRINT
```

Also see LPOS Function, Section 7.82.

7.108 PRESET STATEMENT

Syntax:

PRESET [STEP](<xcoordinate>,<ycoordinate>) [,<color>]

<xcoordinate> and <ycoordinate> specify the pixel that is to be set.

<color> is the color number that is to be used for the specified point.

The STEP option, when used, indicates the given x and y coordinates will be relative, not absolute. That means the x and y are distances from the most recent cursor location, not distances from the (0,0) screen coordinate.

Purpose:

To draw a specified point on the screen. PRESET works exactly like PSET except that if the <color> is not specified, the background color is selected.

Remarks:

If a coordinate is outside the current viewport, no action is taken, nor is an error message given.

Coordinates can be shown as absolutes, as in the above syntax, or the STEP option can be used to reference a point relative to the most recent point used. For example, if the most recent point referenced were (10,10), STEP (10,5) would reference the point at (20,15).

Example:

```
 5 REM DRAW A LINE FROM (0,0) TO (100, 100)
10 FOR i=0 TO 100
20 PRESET (i,i), 1
30 NEXT i
35 REM NOW ERASE THAT LINE
40 FOR i=0 TO 100
50 PRESET STEP (-1,-1)
60 NEXT i
```

This example draws a line from (0,0) to (100,100) and then erases that line by overwriting it with the background color.

7.109 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 on the screen. The expressions in the list may be numeric and/or string expressions. (String literals must be enclosed in quotation marks.)

A question mark (?) can be used as a form of shorthand by the user. It will be interpreted as the word "PRINT", and will appear as "PRINT" in subsequent listings.

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. SLE 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 according to instructions. 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 wider than the screen width, SLE 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 .000000000000001 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. (Remember, a number is always followed by a space, and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

7.110 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 "\   \";A4;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-justified, and the exponent is adjusted. Unless a leading + or trailing + or – is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
2.35D+02
PRINT USING ".####^^^^-";-888888
-.8839E+06
PRINT USING "+.##^^^^";123
+.12E+03
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

% 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.

7.111 PRINT# AND 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 7.110. 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 7.158.

7.112 PSET STATEMENT

Syntax:

PSET [STEP](<xcoordinate>,<ycoordinate>)
[,<color>]

(<xcoordinate> and <ycoordinate> specify the point on the screen to be colored.

<color> is the number of the color to be used.

The STEP option, when used, indicates the given x and y coordinates will be relative, not absolute. That means the x and y are distances from the most recent cursor location, not distances from the (0,0) screen coordinate.
Purpose:

To draw a specified point on the screen.

Remarks:

When SLE GW-BASIC scans coordinate values, it will allow them to be beyond the edge of the screen (size of the screen is dependent on the particular machine being used, and can be adjusted with the WIDTH statement). However, values outside the integer range -32768 to 32767 will cause an "Overflow" error.

Coordinates can be shown as offsets by using the STEP option to reference a point relative to the most recent point used. The syntax of the STEP option is:

STEP (<xoffset>,<yoffset>)

For example, if the most recent point referenced were (0,0), PSET STEP (10,0) would reference a point at offset 10 from x and offset 0 from y.

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

PSET allows the <color> to be left off the command line. If it is omitted, the default is the foreground color.

Example:

```
 5 REM DRAW A LINE FROM (0,0) TO (100,100)
10 FOR I=0 TO 100
20   PSET (I,I)
30 NEXT I
35 REM NOW ERASE THAT LINE
40 FOR I=0 TO 100
50   PSET STEP (-1,-1),0
60 NEXT I
```

This example draws a line from (0,0) to (100,100) and then erases that line by writing over it with the background color.

7.113 PUT STATEMENT — FILE I/O

Syntax:

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 or GET). The largest possible record number is 16,777,215. The smallest record number is 1.

The GET and PUT statements allow fixed-length input and output for SLE 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.

NOTE

LSET, RSET, 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#, SLE 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.

For details on file I/O, see Chapter 5, "Working with Files and Devices."

Example:

```
100 PUT   1, A$, B$, C$
```

7.114 PUT STATEMENT — GRAPHICS

Syntax:

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

used with

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

(x1,y1) in the PUT statement specifies the point where a stored image is to be displayed on the screen. The specified point is the coordinate of the top left corner of the image. If the image to be transferred is too large to fit in the current viewport, an "Illegal function call" error will result.

<action verb> is one of: PSET, PRESET, AND, OR, XOR.

PSET transfers the data point by point onto the screen. Each point has the exact color attribute it had when it was taken from the screen with a GET.

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

AND is used when the image is to be transferred over an existing image on the screen. The resulting image is the product of the logical AND expression; points that had the same color in both the existing image and the PUT image will remain the same color, points that do not have the same color in both the existing image and the PUT image will not.

OR is used to superimpose the image onto an existing image.

XOR is a special mode often used for animation. It causes the points on the screen to be inverted where a point exists in the array image. This behavior is exactly like that of the cursor. When an image is PUT against a complex background twice, the background is restored unchanged. This allows a user to move an object around the screen without erasing the background.

The default <action verb> is XOR.

Purpose:

The GET and PUT statements are used together to transfer graphic images to and from the screen.

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

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

The <action verb> specifies the interaction between the stored image and the one already on the screen.

Remarks:

One of the most useful things that can be done with GET and PUT is animation. Animation is performed as follows:

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

Movement done this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1 and by making sure that there is enough time delay between 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. The idea is to leave a border around the image when it is first gotten that is as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points left by the previous PUT. This method may be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

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

7.115 RANDOMIZE STATEMENT

Syntax:

RANDOMIZE [<expression>]

Purpose:

To reseed the random number generator.

Remarks:

If <expression> is omitted, SLE GW-BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If expression is a variable, the value of that variable is used to seed the random numbers.

If <expression> is the word "TIMER" then the TIMER function is used to pass a random number seed.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the 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 .4845668 .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.

7.116 READ STATEMENT

Syntax:

READ <list of variables>

Purpose:

To read values from a DATA statement and assign them to variables. (See "DATA Statement," Section 7.27.)

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 7.120).

Example 1:

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

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

Example 2:

```
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z$
30 DATA "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.

7.117 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.

CAUTION

Do not use the single quotation form of the REM statement 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
  .
  .
  .
```

7.118 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, ELSE, ON...GOTO, ON...GOSUB, RESTORE, RESUME, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number 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.

7.119 RESET COMMAND

Syntax:

```
RESET
```

Purpose:

To close all files.

Remarks:

RESET closes all open files and forces all blocks in memory to be written to disk. Thus, if the machine loses power, all files will be properly updated.

All files must be closed before a disk is removed from its drive.

Example:

```
998 RESET
999 END
```

7.120 RESTORE STATEMENT

Syntax:

RESTORE [<line number>]

Purpose:

To allow DATA statements to be reread from a specified line.

Remarks:

After a RESTORE statement without a specified line number is executed, the next READ statement accesses the first item in the first DATA statement in the program.

Is <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
 .
 .
 .
```

7.121 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(ERL=90) THEN PRINT
"TRY AGAIN":RESUME 80
  .
  .
  .
```

7.122 RETURN STATEMENT

See GOSUB...RETURN Statements, Section 7.54.

7.123 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 7.69 and 7.87, respectively.

7.124 RMDIR STATEMENT

Syntax:

RMDIR<pathname>

Purpose:

To remove an existing directory

Remarks:

PATHNAME is the name of the directory which is to be deleted. RMDIR works exactly like the SLE MS-DOS command RMDIR. The PATH-NAME must be a string of less than 128 characters.

The PATHNAME to be removed must be empty of any files except the working directory('.') and the parent directory('..') or else a "Path not found" or a "Path/File Access error" is given.

Example:

RMDIR "\SALES"

In this statement, the SALES directory on the current drive is to be removed.

7.125 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 7.115). 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 I
```

might yield

```
 24  30  31  51  5
```

NOTE

The values produced by the RND function may vary with different implementations of SLE GW-BASIC.

7.126 RUN STATEMENT/COMMAND

Syntaxes:

RUN [<line number>]
RUN <filespec>[,R]

Purpose:

To execute the program currently in memory, or to load a file into memory and run it.

Remarks:

For a program currently in memory, if <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. SLE GW-BASIC always returns to command level after a RUN statement is executed.

For running a program not in memory, the <filespec> is an optional device specification followed by a filename or pathname that conforms to SLE MS-DOS's rules for filenames. BASIC appends the default filename extension .BAS if the user specifies no extensions, and the file has been saved to the disk.

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

7.127 SAVE COMMAND

Syntax:

SAVE <filespec>[{,A|,P}]

Purpose:

To save a program file.

Remarks:

For saving a program in memory, the <filespec> is an optional device specification followed by a file name or pathname that conforms to SLE MS-DOS's rules for filenames. BASIC appends the default filename extension ".BAS" if the user specifies no extensions, and the file has been saved to the disk.

The A option saves the file in ASCII format. If the A option is not specified, SLE GW-BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some actions require that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as TYPE 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.BAS as a protected file which cannot be altered.

7.128 SCREEN STATEMENT

Syntax:

SCREEN [*mode*] [,[*burst*] [,[*apage*] [,*vpage*]]]

mode is a numeric expression resulting in an integer value of 0,1,2 or 3. Valid modes are:

0 Text mode at current width (40 or 80).

1 graphics mode (320x200).

2 graphics mode (640x200)

burst is a numeric expression resulting in a true or false value. It enables color. In text mode (*mode*=0), a false (zero) value disables color (black and white images only) and a true (non-zero) value enables color (allows color images). In graphics mode a true (non-zero) value will disable color, and a false (zero) value will enable color.

apage (active page) is an integer expression in the range 0 to 7 in text mode or 0 to 1 in graphics mode. It selects the page to be written to by output statements to the screen. If omitted, *apage* defaults to the previous *apage*.

vpage (visual page) selects which page is to be displayed on the screen, in the same way as *apage* above. The visual page may be different than the active page. If omitted, *vpage* defaults to *apage*.

Purpose:

To set the specifications for the display screen. The specifications may vary according to individual machines.

Remarks:

If all parameters are valid, the new screen mode is stored, the screen is erased, the foreground color is set to white, and the background and border colors are set to black.

If the new screen mode is the same as the previous mode, nothing is changed.

If the mode is text, and only *apage* and *vpage* are specified, the effect is that of changing display pages for viewing. Initially, both active and visual pages default to 0 (zero). By manipulating active and visual pages, you can display one page while building another. Then you can switch visual pages instantaneously.

NOTE

There is only one cursor shared between all the pages. If you are going to switch active pages back and forth, you should save the cursor position on the current active page (using POS(0) and CSRLIN), before changing to another active page. Then when you return to the original page, you can restore the cursor position using the LOCATE statement.

Any parameter may be omitted. Omitted parameters, except *vpage*, assume the old value.

Any values entered outside of the ranges indicated will result in an "Illegal function call" error. Previous values are retained.

Example:

```
10 SCREEN 0,1,0,0
```

Selects text mode with color, and sets active and visual page to 0.

```
20 SCREEN ,,1,2
```

Mode and color burst remain unchanged. Active page is set to 1 and display page to 2.

7.129 SCREEN FUNCTION

Syntax:

SCREEN(*row*,*col*[,*z*])

row is a numeric expression in the range 1 to 25.

col is a numeric expression in the range 1 to 40 or 1 to 80 depending upon the WIDTH setting.

z is a numeric expression which evaluates to a true or false value. *z* is only valid in text mode.

Purpose:

To read a character or its color from a specified screen location.

Remarks:

The ordinate of the character at the specified coordinates is stored in the numeric variable. If the optional parameter z is given and is non-zero, the color attribute for the character is returned instead.

In text mode, if z is included and nonzero, the color attribute for the character is returned instead of the ASCII value of the character. The color attribute is a number from 0 to 255. This number, x, is used as follows:

(x MOD 16) is the foreground attribute.

(((x - foreground) / 16) MOD 128) is the background attribute.

If x > 127 then the character is blinking. If x < 127 then the character is not blinking.

In graphics mode, if the specified location contains graphics information, the SCREEN function returns 0.

Example:

```
100 x = SCREEN (10,10)
```

If the character at (10,10) is A, then the function would return 65, the ASCII code for A.

```
100 x = SCREEN (1,1,1)
```

Returns the color attribute of the character in the upper left corner of the screen.

7.130 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 is positive.

7.131 SHELL STATEMENT

Syntax:

SHELL [<command-string>]

Purpose:

To exit the BASIC program, run a COM or EXE or BAT program, or a built-in DOS function such as DIR or TYPE, and return to the BASIC program at the line after the SHELL statement.

Remarks:

A COM, EXE, or BAT program or DOS function which runs under the SHELL statement is called a "Child process." Child processes are executed by SHELL loading and running a copy of COMMAND with the "/C" switch. By using COMMAND in this way, command line parameters are passed to the child. Standard input and output may be redirected, and built-in commands such as DIR, PATH, and SORT may be executed.

The <command-string> must be a valid string expression containing the name of a program to run and (optionally) command arguments.

The program name in <command-string> may have any extension you wish. If no extension is supplied, COMMAND will look for a .COM file, then a .EXE file, and finally, a .BAT file. If COMMAND is not found, SHELL will issue a "File not found" error. No error is generated if COMMAND cannot find the file specified in <command-string>.

Any text separated from the program name by at least one blank will be processed by COMMAND as program parameters.

BASIC remains in memory while the child process is running. When the child finishes, BASIC continues.

SHELL with no <command-string> will give you a new COMMAND shell. You may now do anything that COMMAND allows. When ready to return to BASIC, enter the DOS command: EXIT.

Examples:

```
SHELL 'get a new COMMAND
A>DIR        {user types DIR to see files}
A>EXIT       }user types EXIT to return to BASIC}
Ok           'now back in BASIC
```

Write some data to be sorted, SHELL sort to sort it, then read the sorted data to write a report.

```
900 OPEN "SORTIN.DAT" FOR OUTPUT AS 1
950 REM ** write data to be sorted
1000 CLOSE 1
1010 SHELL "SORT SORTIN.DAT SORTOUT.DAT"
1020 OPEN "SORTOUT.DAT" FOR INPUT AS 1
1030 REM ** Process the sorted data

10 SHELL "DIR | SORT FILES.
20 OPEN "FILES." FOR INPUT AS 1
```

Also see BASIC and Child Processes, Section 5.7.

7.132 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 7.22.

7.133 SPACE$ FUNCTION

Syntax:

SPACE$(I)

Purpose:

To return a string of spaces of length I.

Remarks:

The expression I is rounded to an integer and must be in the range 0 to 255.

Example:

```
10 FOR I=1 TO 5
20    X$=SPACE$(I)
30    PRINT X$;I
40 NEXT I
```

will yield

```
 1
  2
   3
    4
     5
```

Also see SPC Function, Section 7.134.

7.134 SPC FUNCTION

Syntax:

SPC(n)

Purpose:

To skip spaces in a PRINT statement. n is the number of spaces to be skipped.

Remarks:

SPC may only be used with PRINT and LPRINT statements. n must be in the range 0 to 255. A';' is assumed to follow the SPC(n) command.

Example:

```
PRINT "OVER" SPC(15) "THERE"
will yield
OVER               THERE
```

Also see SPACE$ Function, Section 7.133.

7.135 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 X
```

will yield

```
10          3.162278
15          3.872984
20          4.472136
25          5
```

7.136 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

The STOP Statement doesn't close files.

SLE GW-BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 7.21).

Example:

```
10 INPUT A,B,C
20 K=A ^ 2*5.3:L=B ^ 3/.26
30 STOP
40 M=C*K+100:PRINT M
```

will yield

```
? 1,2,3
BREAK IN 30
PRINT L
 30.76923
CONT
 115.9
```

7.137 STR$ FUNCTION

Syntax:

STR$(n)

Purpose:

To return a string representation of the value of n.

Example:

```
 5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N))-1 GOSUB 30,100,200,300,400,500
 .
 .
 .
```

Also see VAL Function, Section 7.149.

7.138 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$.

Examples:

```
10 DASH$ = STRING$(10,45)
20 PRINT DASH$;"MONTHLY REPORT";DASH$
```

will yield

```
----------MONTHLY REPORT----------
```

```
10 LET A$ = "HOUSTON"
20 LET X$ = STRING$(8,A$)
30 PRINT X$
```

will yield

```
HHHHHHHH
```

7.139 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
```

7.140 SYSTEM COMMAND

Syntax:

SYSTEM

Purpose:

To close all open files and return control to the operating system.

Remarks:

When a SYSTEM command is executed, all files are closed, and BASIC performs an exit to the operating system.

7.141 TAB FUNCTION

Syntax:

TAB(B)

Purpose:

To move the print position to B.

Remarks:

If the current print position is already beyond space B, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. B 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
```

7.142 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
```

7.143 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.

7.144 TIME$ FUNCTION

Syntax:

TIME$

Purpose:

To retrieve the current time. (To set the time, use the TIME$ statement, described in Section 7.143.)

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.

7.145 TIMER FUNCTION

Syntax:

TIMER

Purpose:

Return a single-precision number representing the number of seconds that have elapsed since midnight.

7.146 TIMER ON, TIMER OFF, TIMER STOP STATEMENTS

Syntax:

```
TIMER ON
TIMER OFF
TIMER STOP
```

Purpose:

TIMER ON enables event trapping during real time. TIMER OFF disables event trapping during real time. TIMER STOP suspends real time event trapping.

Remarks:

The TIMER ON statement enables real time event trapping by an ON TIMER statement (see "ON TIMER Statement," Section 7.97). While trapping is enabled with the ON TIMER statement, SLE GW-BASIC checks between every statement to see if the timer has reached the specified level. If it has, the ON TIMER statement is executed.

TIMER OFF disables the event trap. If an event takes place, it is not remembered if a subsequent TIMER ON is used.

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

Also see ON TIMER Statement, Section 7.97.

7.147 TRON/TROFF STATEMENTS/COMMANDS

Syntax:

TRON

TROFF

Purpose:

To trace the execution of program statements.

Remarks:

As an aid in debugging, the TRON statement may be executed in either direct or indirect mode. With TRON in operation, each line number of the program is printed on the screen 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 J
70 END
```

will yield

```
[10][20][30][40] 1  10  20
[50][60][30][40] 2  20  30
[50][60][70]
```

7.148 USR FUNCTION

Syntax:

USR[<digit>][(<argument>)]

where <digit> specifies which USR routine is being called. See the DEF USR statement, Section 7.33, 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:

If a segment other than the default segment (data segment) is to be used, a DEF SEG statement must be executed prior to a USR function call. The address given in the DEF SEG statement determines the segment address of the subroutine.

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

Example:

```
100 DEF SEG=&H8000
110 DEF USR0=0
120 X=5
130 Y = USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the value must be consistent with the argument passed. This setup of the string space differs from that of the interpreter.

When the USR function call is made, register AL contains a value that specifies the type of argument that was given. The value in AL may be one of the following:

VALUE IN AL	TYPE OF ARGUMENT
2	Two-byte integer (two's complement)
3	String
4	Single precision floating-point number
8	Double precision floating-point number

If the argument is a number and not a string, the value of the argument is placed in the Floating Point Accumulator (FAC), which is an eight-byte area in BASIC's data space. In this case, the BX register contains the offset within the BASIC data space to the fifth byte of the eight-byte FAC. For the following examples, assume that the FAC is in bytes hex 49F through hex 4A6; that is, BX contains hex 4A3:

If the argument is an integer:

- Hex 4A4 contains the upper 8 bits of the argument.
- Hex 4A3 contains the lower 8 bits of the argument.

If the argument is a single-precision number:

- Hex 4A6 contains the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa. Hex 4A5 contains the highest 7 bits of the mantissa with the leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive; 1 = negative).
- Hex 4A4 contains the middle 8 bits of the mantissa.
- Hex 4A3 contains the lowest 8 bits of the mantissa.

If the argument is a double-precision number:

- Hex 4A3 through hex 4A6 are the same as described under single-precision floating-point number in the preceding paragraph.
- Hex 49F through Hex 4A2 contain four more bytes of the mantissa (hex 49F contains the lowest 8 bits).

If the argument is a string, the DX register points to 3 bytes which, as a unit, are called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255 characters). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in the SLE GW-BASIC data segment.

CAUTION

> If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy the program this way.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

SLE GW-BASIC has extended the USR function interface to allow calls to MAKINT and FRCINT. This allows access to these routines without giving their absolute addresses. The address ES:BP is used as an indirect far pointer to the routines FRCINT and MAKINT.

To call FRCINT from a USR routine use CALL DWORD ES:[BP]. To call MAKINT from a USR routine use CALL DWORD ES:[BP+4].

Example:

```
110 DEF USR0=&H8000 'Assumes decimal argument / M:32767
120 X= 5
130 Y = USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with that of the argument used.

7.149 VAL FUNCTION

Syntax:

VAL(<string>)

<string> must be a numeric character stored as a string.

Purpose:

To return the numeric value of string <string>. 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$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699
   THEN PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815
   THEN PRINT NAME$ TAB(25) "LONG BEACH"
  .
  .
  .
```

See the STR$ function, Section 7.137, for details on numeric-to-string conversion.

7.150 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>. The variable must have been defined prior to the execution of the VARPTR function. Otherwise an "Illegal function call" error results. Variables are defined by executing any reference to the variable.

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," Section 6.1 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

The second format returns the starting address of the file control block for the specified file. This is not the same as the DOS file control block.

Example:

This example reads the first byte in the buffer of a random file:

```
10 OPEN "DATA.FIL" AS #1
20 GET #1
30 'get address of control block
40 FCBADR = VARPTR(#1)
50 'figure address of data buffer
60 DATADR = FCBADR+51
70 'get first byte in data buffer
80 A% = PEEK(DATADR)
```

7.151 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 in a form that is compatible for programs that may later be compiled.

Remarks:

NOTE

All simple variables should be assigned before calling VARPTR$ for an array element, because arrays are relocated whenever a new simple variable is assigned.

VARPTR$ returns a 3-byte string in the form:

Byte 0	Byte 1	Byte 2
type	low byte of variable address	high byte of variable address

type indicates the variable type:

2	integer
3	string
4	single-precision
8	double-precision

The returned value is the same as:

CHR$(type)+MKI$(VARPTR(variable))

You can use VARPTR$ to indicate a variable name in the command string for DRAW.

Example

(Method 1)

```
10 DRAW "E15;A$;"
```

(Method 2)

```
10 DRAW "E15"+VARPTR$(A$)
```

7.152 VIEW STATEMENT

Syntax:

VIEW [[SCREEN] [(Vx1,Vy1)–(Vx2,Vy2) [,[<color>]
[,[<border>]]]]]

Purpose:

To define screen limits for graphics activity.

Remarks:

VIEW defines a "Physical Viewport" limit from Vx1,Vy1 (upper left x,y coordinates) to Vx2,Vy2 (lower right x,y coordinates). The x and y coordinates must be within the physical bounds of the screen. The physical viewport defines the rectangle within the screen into which graphics may be mapped.

RUN, and RUN, SCREEN and VIEW with no arguments, define the entire screen as the viewport.

The <color> attribute lets you fill the defined viewport with color. If <color> is omitted, the viewport is not filled. <color> is an integer expression that chooses an attribute from the attribute range for the current screen mode. In medium resolution, the color is the current one for that attribute as defined by the COLOR statement. Four attributes (0-3) are available in medium resolution; in high resolution, two attributes (0-1) are available. Zero (0) is always the attribute for the background. The default foreground attribute is always the maximum attribute for that screen mode: 3 in medium resolution; 1 in high resolution.

The <border> attribute allows the user to draw a line surrounding the viewport if space for a border is available. If border is omitted, no border is drawn.

The [SCREEN] option dictates that the x and y coordinates are absolute to the screen, not relative to the border of the physical viewport, and only graphics within the viewport will be plotted.

Examples:

For the form:

VIEW (Vx1,Vy1)–(Vx2,Vy2)

all points plotted are relative to the viewport. That is, Vx1 and Vy1 are added to the x and y coordinates before putting the point down on the screen.

If:

VIEW (10,10)–(200,100)

were executed, then the point set down by the statement PSET (0,0) would actually be at the physical screen location 10,10.

For the form:

VIEW SCREEN (Vx1,Vy1)–(Vx2,Vy2)

all coordinates are screen absolute rather than viewport relative.

If:

VIEW SCREEN (10,10)–(200,100)

were executed, then the point set down by the statement PSET (0,0),3 would actually not appear because 0,0 is outside of the viewport. PSET (10,10),3 is within the viewport, and places the point in the upper-left hand corner of the viewport.

A number of VIEW statements may be executed. If the newly described viewport is not wholly within the previous viewport, the screen can be re-initialized with the VIEW statement. Then the new viewport may be stated. If the new viewport is entirely within the previous one, as in the following example, the intermediate VIEW statement isn't necessary. This example opens three viewports, each smaller than the previous one. In each case, a line that is defined to go beyond the borders is programmed, but appears only within the viewport border.

```
260 CLS
280 VIEW: REM ** Make the viewport the entire screen.
300 VIEW (10,10) - (300,180),,1
320    CLS
340    LINE (0,0) - (310,190),1
360    LOCATE 1,11: PRINT "A big viewport"
380 VIEW SCREEN (50,50)-(250,150),,1
400    CLS:REM** Note, CLS clears only viewport
420    LINE (300,0)-(0,199),1
440    LOCATE 9,9: PRINT "A medium viewport"
460 VIEW SCREEN (80,80)-(200,125),,1
480    CLS
500    CIRCLE (150,100),20,1
520    LOCATE 11,9: PRINT "A small viewport"
```

7.153 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, SLE 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
```

7.154 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. SLE GW-BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
 90 'BUBBLE SORT ARRAY A$ WHICH HAS J ELEMENTS.
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.

7.155 WIDTH STATEMENT

Syntax 1:

WIDTH [LPRINT]<size>

Syntax 2:

WIDTH <file number>,<size>

Syntax 3:

WIDTH <device>,<size>

<size> is a numeric expression in the range 0 to 255. It specifies the width of the printed line. The default width is 72 characters. If LPRINT is omitted, <size> can only be 40 or 80.

If <integer expression> is 255, the line width is "infinite"; that is, SLE GW-BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

<file number> is a numeric expression in the range 1 to 15. This is the number of the file that is open.

<device> is a string expression indicating the device that is to be used.

Purpose:

To set the printed line width in number of characters for the screen or line printer.

Remarks:

Syntax 1: If the LPRINT option is omitted, the line width is set at the screen. If LPRINT is included, the line width is set at the line printer.

The WIDTH statement may cause the screen to be cleared.

Syntax 2: With Syntax 2, if the file is open, the width is immediately changed to the specified size. This allows the width to be changed while the file is open.

Syntax 3: With Syntax 3, the default line width for the specified device is set to <size>. The line widths of currently open files are not modified. A subsequent OPEN <filespec> FOR OUTPUT AS #n will use the specified value for the width initially.

Example:

```
10 WIDTH "LPT1:", 5
20 OPEN "LPT1:" FOR OUTPUT AS 1
30 PRINT 1, "1234567890"
35 PRINT 1
40 WIDTH 1, 6
50 PRINT 1, "1234567890"
RUN
```

will yield

```
12345
67890

123456
7890
```

7.156 WINDOW STATEMENT

Syntax:

WINDOW [[SCREEN] (Wx1,Wy1)-(Wx2,Wy2)]

where (Wx1,Wy1)-(Wx2,Wy2) are the world coordinates specified by the programmer to define the coordinates of the lower left and upper right screen border.

SCREEN inverts the y axis of the world coordinates so that screen coordinates coincide with the traditional Cartesian arrangement: x increases left to right, and y decreases top to bottom.

Purpose:

To define the logical dimensions of the current viewport.

Remarks:

WINDOW allows the user to redefine the screen border coordinates.

WINDOW allows the user to draw lines, graphs, or objects in space not bounded by the physical dimensions of the screen. This is done by using programmer-defined coordinates called "World coordinates". When the programmer has redefined the screen, graphics can be drawn within a customized mapping system.

BASIC converts world coordinates into physical coordinates for subsequent display within the current viewport. To make this transformation from world space to the physical space of the viewing surface (screen), one must know what portion of the (floating point) world coordinate space contains the information to be displayed. This rectangular region in world coordinate space is called a Window.

RUN, or WINDOW with no arguments, disables "Window" transformation.

The WINDOW SCREEN variant inverts the normal Cartesian direction of the y coordinate. Consider the following:

In the default, a section of the screen appears as:

```
0,0                              : :    50,0                        100,0
                                 : :
                                 \/ y increases
                                         100,0
0,100                            50,100                           100,100
```

Now execute:

WINDOW (-1,-1)-(1,1)

and the screen appears as:

```
-1,1                                 0,1                            1,1
                                 /\ y increases
                                 : :
                                 : :
                                           0,0
                                 : :
                                 \/ y decreases
-1,-1                                0,-1                           1,-1
```

If the variant:

WINDOW SCREEN (-1,-1)-(1,1)

is executed then the screen appears as:

```
-1,-1                                0,-1                           1,-1
                                 /\ y decreases
                                 : :
                                 : :        0,0
                                 : :
                                 \/ y increases
-1,1                                 0,1                            1,1
```

The following example illustrates two lines with the same endpoint coordinates. The first is drawn on the default screen, and the second is on a redefined window.

```
200 LINE (100,100) - (150,150), 1
220 LOCATE 2,20:PRINT "The line on the default screen"
240 WINDOW SCREEN (100,100) - (200,200)
260 LINE (100,100) - (150,150), 1
280 LOCATE 8,18:PRINT"& the same line on a redefined window"
```

7.157 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, SLE GW-BASIC inserts a carriage return/linefeed.

WRITE outputs numeric values using the same format as the PRINT statement. (See Section 7.109.)

Example:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
```

will yield

```
80, 90,"THAT'S ALL"
```

7.158 **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 7.98). 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$,B$
```

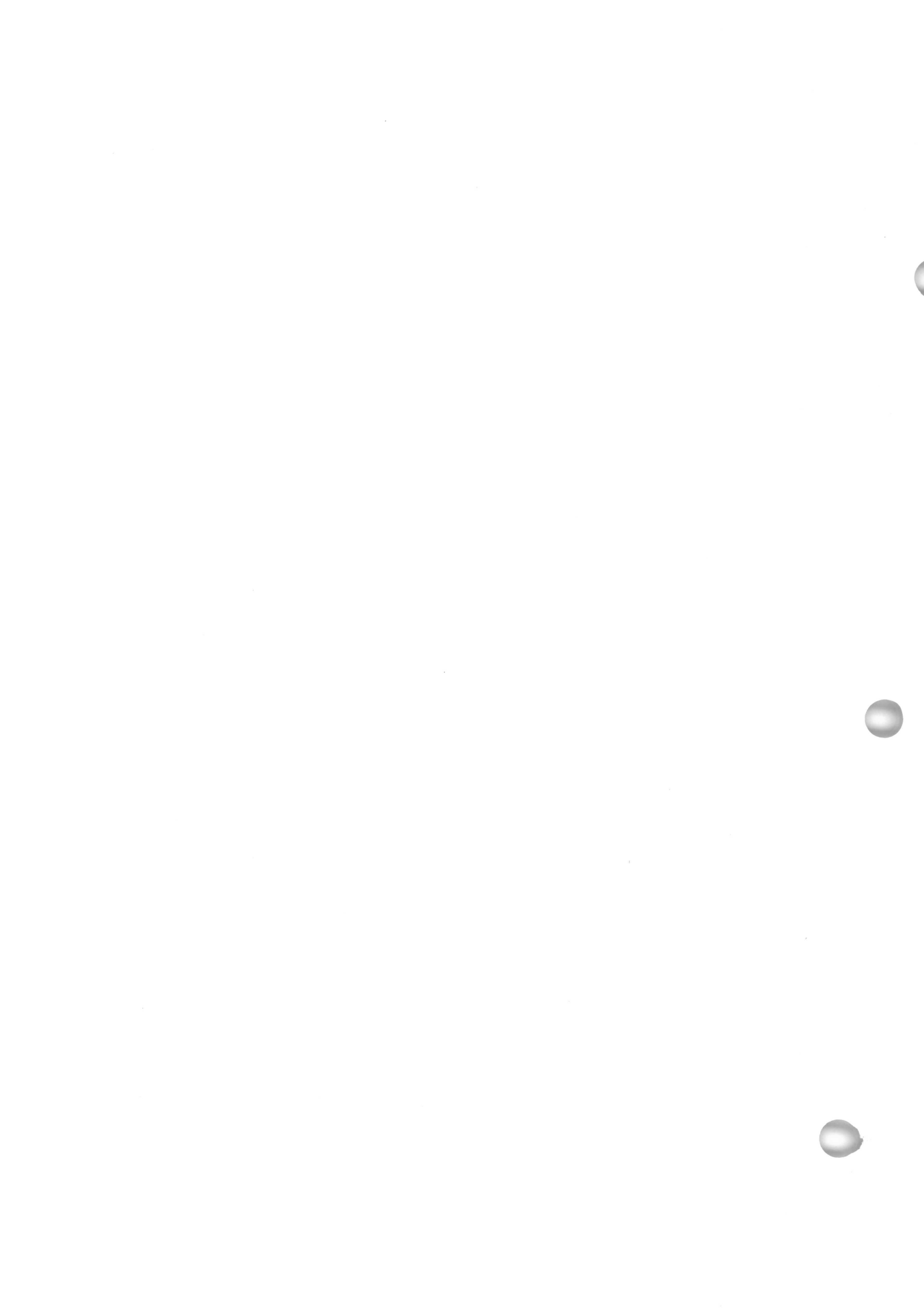
writes the following image to disk:

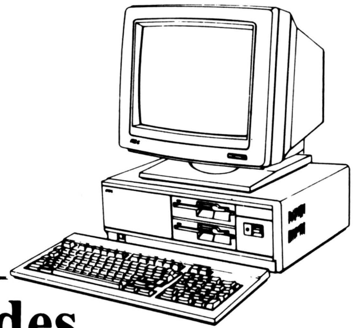
```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as

```
INPUT#1,Z$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.





Appendix A

ASCII Character Codes

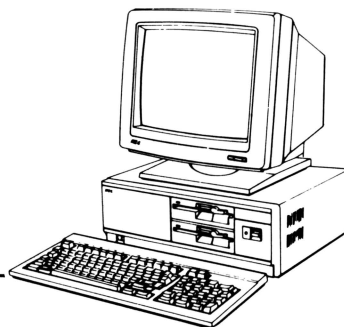
ASCII VALUE	CHARACTER	CONTROL CHARACTER	ASCII VALUE	CHARACTER
000	(null)	NUL	032*	(space)
001	☺	SOH	033	!
002	●	STX	034	"
003	♥	ETX	035	#
004	♦	EOT	036	S
005	♣	ENQ	037	%
006	♠	ACK	038	&
007	(null)	BEL	039	'
008*	◘	BS	040	(
009*	○	HT	041	)
010*	◙	LF	042	*
011*	♂	VT	043	+
012*	♀	FF	044	,
013*	♪	CR	045	–
014	♫	SO	046	.
015	☼	SI	047	/
016	►	DLE	048	0
017	◄	DC1	049	1
018	↕	DC2	050	2
019	!!	DC3	051	3
020	π	DC4	052	4
021	§	NAK	053	5
022	▬	SYN	054	6
023	↨	ETB	055	7
024	↑	CAN	056	8
025	↓	EM	057	9
026	→	SUB	058	:
027	←	ESC	059	;
028*	∟	FS	060	<
029*	↔	GS	061	=
030*	▲	RS	062	>
031*	▼	US	063	?

* These ASCII values do not display a displayable character but are interpreted as control characters. One way to get the displayable character for one of these particular ASCII values in GW-BASIC, is to 'POKE' that value into the appropriate location in video RAM.

ASCII VALUE	CHARACTER	ASCII VALUE	CHARACTER
064	@	104	h
065	A	105	i
066	B	106	j
067	C	107	k
068	D	108	l
069	E	109	m
070	F	110	n
071	G	111	o
072	H	112	p
073	I	113	q
074	J	114	r
075	K	115	s
076	L	116	t
077	M	117	u
078	N	118	v
079	O	119	w
080	P	120	x
081	Q	121	y
082	R	122	z
083	S	123	{
084	T	124	|
085	U	125	}
086	V	126	~
087	W	127	⌂
088	X	128	Ç
089	Y	129	ü
090	Z	130	é
091	[	131	â
092	\	132	ä
093	]	133	à
094	^	134	å
095	_	135	ç
096	`	136	ê
097	a	137	ë
098	b	138	è
099	c	139	ï
100	d	140	î
101	e	141	ì
102	f	142	Ä
103	g	143	Å

ASCII VALUE	CHARACTER	ASCII VALUE	CHARACTER
144	É	184	╕
145	æ	185	╣
146	Æ	186	║
147	ô	187	╗
148	ö	188	╝
149	ò	189	╜
150	û	190	╛
151	ù	191	┐
152	ÿ	192	└
153	Ö	193	┴
154	Ü	194	┬
155	¢	195	├
156	£	196	─
157	¥	197	┼
158	Pts	198	╞
159	ƒ	199	╟
160	á	200	╚
161	í	201	╔
162	ó	202	╩
163	ú	203	╦
164	ñ	204	╠
165	Ñ	205	═
166	ª	206	╬
167	º	207	╧
168	¿	208	╨
169	⌐	209	╤
170	¬	210	╥
171	½	211	╙
172	¼	212	╘
173	¡	213	╒
174	«	214	╓
175	»	215	╫
176	░	216	╪
177	▒	217	┘
178	▓	218	┌
179	│	219	█
180	┤	220	▄
181	╡	221	▌
182	╢	222	▐
183	╖	223	▀

ASCII VALUE	CHARACTER	ASCII VALUE	CHARACTER
224	α	240	≡
225	β	241	±
226	Γ	242	≥
227	π	243	≤
228	Σ	244	⌠
229	σ	245	⌡
230	μ	246	÷
231	τ	247	≈
232	Φ	248	○
233	Θ	249	●
234	Ω	250	°
235	δ	251	√
236	∞	252	η
237	∅	253	·
238	Σ	254	■
239	∩	255	(blank 'FF')



Appendix B

Error Codes and Error Messages

NUMBER	MESSAGE
1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as an unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.). With SLE GW-BASIC, the incorrect line will be part of a DATA statement. SLE GW-BASIC Interpreter automatically enters edit mode at the line that caused the error.
3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.

NUMBER	MESSAGE
5	Illegal function call A parameter that is out of range is passed to a math or string function. An 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 noninteger 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$, INT, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO. 7. A negative record number used with GET or PUT.
6	Overflow The result of a calculation is too large to be represented in SLE GW-BASIC number format. If underflow occurs, the result is zero and execution continues without an error.
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.
8	Undefined line A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.
9	Subscript out of range An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.

NUMBER	MESSAGE
10	Duplicate definition Two DIM statements are given for the same array; or, a DIM statement is given for an array after the default dimension of 10 has been established for that array.
11	Division by zero A division by zero is encountered in an expression; or, the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.
12	Illegal direct A statement that is illegal in direct mode is entered as a direct mode command.
13	Type mismatch A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.
14	Out of string space String variables have caused BASIC to exceed the amount of free memory remaining. SLE GW-BASIC will allocate string space dynamically, until it runs out of memory.
15	String too long An attempt is made to create a string more than 255 characters long.
16	String formula too complex A string expression is too long or too complex. The expression should be broken into smaller expressions.

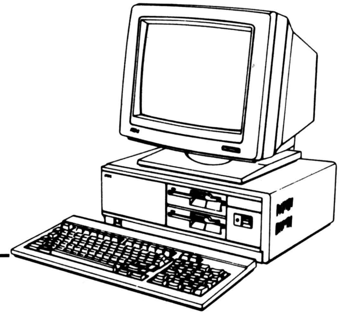
NUMBER	MESSAGE
17	Can't continue An attempt is made to continue a program that: 1. Has halted due to an error. 2. Has been modified during a break in execution. 3. Does not exist.
18	Undefined user function A USR function is called before the function definition (DEF USRn statement) is given.
19	No RESUME An error handling routine is entered but contains no RESUME statement.
20	RESUME without error A RESUME statement is encountered before an error handling routine is entered.
21	Unprintable error An error message is not available for the error condition that exists.
22	Missing operand An expression contains an operator with no operand following it.
23	Line buffer overflow An attempt has been made to input a line that has too many characters.
24	Device timeout The device you have specified is not available at this time.
25	Device fault An incorrect device designation has been entered.

NUMBER	MESSAGE
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.
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 SLE GW-BASIC. Report to Microsoft the conditions under which the message appeared.
52	Bad file number A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

NUMBER	MESSAGE
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.
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. 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.

NUMBER	MESSAGE
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 an ASCII-format file. The LOAD is terminated.
67	Too many files An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.
68	Device Unavailable The device that has been specified is not available at this time.
69	Communications buffer overflow Not enough space has been reserved for communications I/O.

NUMBER	MESSAGE
70	Disk write protected The disk has a write protect tab intact, or is a disk that cannot be written to.
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 A hardware or disk problem occurred while the disk was being written to or read from. For example, the disk may be damaged or the disk drive may not be working properly.
74	Rename across disks An attempt was made to rename a file with a new drive designation. This is not allowed.
75	Path/file access error During an OPEN, MKDIR, CHDIR, or RMDIR operation, SLE MS-DOS was unable to make a correct Path to File name connection. The operation is not completed.
76	Path not Found During an OPEN, MKDIR, CHDIR, or RMDIR operation, SLE MS-DOS was unable to find the path specified. The operation is not completed.
**	Can't continue after SHELL No error number. Upon returning from a Child process, the SHELL statement discovers that there is not enough memory for BASIC to continue. BASIC closes any open files and exits to SLE MS-DOS.



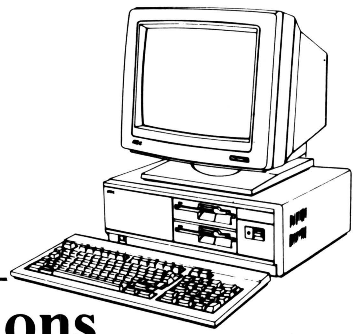
Appendix C

SLE GW-BASIC Reserved Words

The following is a list of reserved words used in SLE GW-BASIC.

ABS
AND
ASC
ATN
AUTO
BEEP
BLOAD
BSAVE
CALL
CALL$
CDBL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COLOR
COM
COMMON
CONT
COS
CSNG
CSRLIN
CVD
CVI
CVS
DATA
DATE$
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
DRAW
EDIT
ELSE
END
ENVIRON
ENVIRON$
EOF
EQV
ERASE
ERDEV
ERDEV$
ERL
ERR
ERROR
END
EXP
FIELD
FILES
FIX
FNxxxxxxxx
FOR
FRE
GET
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT#
INPUT$
INSTR
INT
IOCTL
IOCTL$
KEY
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF

LOG	POKE	STICK
LPOS	POS	STOP
LPRINT	PRESET	STR$
LSET	PRINT	STRIG
MERGE	PRINT#	STRING$
MID$	PSET	SWAP
MKD$	PUT	SYSTEM
MKI$	RANDOMIZE	TAB
MKS$	READ	TAN
MKDIR	REM	THEN
MOD	RENUM	TIME$
MOTOR	RESET	TIMER
NAME	RESTORE	TO
NEW	RESUME	TROFF
NEXT	RETURN	TRON
NOT	RIGHT$	USING
OCT$	RMDIR	USR
OFF	RND	VAL
ON	RSET	VARPTR
OPEN	RUN	VARPTR$
OPTION	SAVE	VIEW
OR	SCREEN	WAIT
OUT	SGN	WEND
PAINT	SHELL	WHILE
PALETTE	SIN	WIDTH
PEEK	SOUND	WINDOW
PEN	SPACE$	WRITE
PLAY	SPC	WRITE#
PMAP	SQR	XOR
POINT	STEP	



Appendix D

Mathematical Functions Not Intrinsic to SLE GW-BASIC

Derived Functions

Functions that are not intrinsic to SLE GW-BASIC may be calculated as follows.

Function	SLE 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(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))

Function	SLE GW-BASIC Equivalent
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/X-1))/2

APC III

NEC
NEC Information Systems, Inc.

USER'S COMMENTS FORM

Document: GW-BASIC Guide for the Software Library Expander

Document No.: 819-150272-000 Rev. 00

Please suggest improvements to this manual.

Please list any errors in this manual. Specify by page.

From:

Name ______

Title ______

Company ______

Address ______

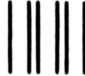
Dealer Name ______

Date: ______

Please cut along this line.

Seal or tape all edges for mailing - do not use staples.

FOLD HERE



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 105 BOXBOROUGH, MA

POSTAGE WILL BE PAID BY ADDRESSEE

NEC Information Systems, Inc.
Dept: Publications
1414 Mass. Ave.
Boxborough, MA 01719

FOLD HERE

Seal or tape all edges for mailing - do not use staples.

Software Documentation Bulletin

819-095000-357 Rev 02

GWBASIC **Model No.: APC-S176**

1.0 INTRODUCTION:

GWBASIC for the NEC APC III is a popular general-purpose interpretive language with easy-to-use, graphics capabilities. In addition, GWBASIC has sophisticated screen handling functions, and support for joysticks and sound.

2.0 CONTENTS OF THE RELEASE:

2.1 Distribution Media

GWBASIC Diskette

2.2 Documentation

GWBASIC Reference Manual

3.0 HARDWARE/SOFTWARE REQUIREMENTS:

3.1 Minimum Hardware Requirements

APC-H101 Monochrome APC-III with 1 FDD with 128 KB memory

3.2 Minimum Software Requirements

MS-DOS 2.11 for APC III

4.0 INSTALLATION PROCEDURES

There are no special installation procedures per se; however, it is highly recommended to format a blank diskette using the FORMAT utility according to the instructions in the MS-DOS User's Guide, pg. 5-33, and then copy the GWBASIC diskette onto the blank formatted diskette using DISKCOPY according to the instructions in the MS-DOS User's Guide, pg. 5-24. The copy that was made should be used and the original should be stored away for safe keeping.

5.0 ENHANCEMENTS:

There are no enhancements to GW-BASIC from the prior release.

6.0 CAUTIONS & RESTRICTIONS:

1. To get a printed copy of what is currently being displayed on the CRT screen when GWBASIC screen modes 1, 2 and 3 are used, it is necessary to load CRTDUMP prior to running GWBASIC (Reference: APC III Supplement to MS-DOS User's Guide p. S4.6). To get a text screen image "dumped" on to the printer type: <Shift> + <print>. To get a graphic screen dump type <shift> + <ctrl> + <print>. The procedure described in GWBasic manual (page 4-9) using <shift> + <print> is for text only! Text dumps work on any supported character printer and graphics dumps only work on P2-3 or P3-3 printer.

2. Variable names over forty characters will result in a "syntax error".

3. Stick function

   When reading joystick B (n=2 or 3) you must first read joystick A (n=0). In addition, if only one joystick is used, you must use A.

4. When using GWBASIC, it is important to remember that the ANSI.SYS Driver (standard input and standard output) is not used. Therefore using a print statement in GWBASIC to issue ESC sequences that are normally interpreted by the ANSI.SYS Driver (eg: cursor positioning and character attribute control) does not work. Instead the normal syntax of GWBASIC must be used to effect cursor addressing and other special attributes normally related to standard input and standard output.

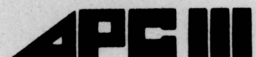
5. When a hard disk APC III system is used other APC III option boards must not use INT3.

6. The Dual Ported RS-232 (APC H156) option board must not use INT0 for both the "COM2:" and "COM3:" ports simultaneously.

7. Since GWBASIC and the GRAPHIC.SYS installable device driver use the same hardware, output to the GRAPH device from GWBASIC will result in incorrect output to the graphics display memory. GWBASIC may be loaded and used independently of the GRAPHIC.SYS installable driver (GRAPH device) as long as output from GWBASIC is not sent to the GRAPH device.

7.0 PROBLEMS FIXED:

1. With the previous version of GW-BASIC, the <CTRL+S> and <CTRL+C> key combinations did not work as described in the GW-BASIC User's Guide page 3-3. This release of GW-BASIC fully supports the <CTRL+S> and <CTRL+C> key combinations as described in the GW-BASIC User's Guide.



CHANGE NOTICE

DATE: September 1985

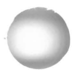
CHANGE NOTICE NUMBER: 819-150328-000
AFFECTED EQUIPMENT: APC III
AFFECTED DOCUMENT: 819-150272-000 GW-BASIC Guide for the Software Library Expander

Insert this Change Notice page just after the copyright page as a means of maintaining an up-to-date record of changes to the manual.

COMMENTS:

CHANGE NOTICE INSTRUCTIONS: Delete and add pages as follows. All changes are indicated by vertical change bars in the outside margin opposite the affected information.

DELETE	ADD
iii, iv	iii, iv
1-3, 1-4, 1-5, 1-6	1-3
2-3, 2-4	2-3, 2-4, 2-5, 2-6
E-1	E-1



Contents

	Page
Chapter 1 Introduction	
1.1 Overview	1-1
1.2 Syntax Notation	1-2
1.3 Resources for Learning BASIC	1-3
Chapter 2 Using SLE GW-BASIC	
2.1 Invoking BASIC	2-1
2.2 Command Line Option Switches	2-2
2.3 Modes of Operation	2-5
2.4 Line Format	2-6
2.5 Active and Visual (Display) Pages	2-6
Chapter 3 Learning the Language	
3.1 Character Set	3-1
3.1.1 Special Characters	3-1
3.1.2 Control Characters	3-3
3.2 Constants	3-4
3.2.1 String and Numeric Constants	3-4
3.2.2 Single/Double Precision Numeric Constants	3-5
3.3 Variables	3-6
3.3.1 Variable Names and Declaration Characters	3-6
3.3.2 Array Variables	3-8
3.3.3 Space Requirements	3-8
3.4 Expressions and Operators	3-9
3.4.1 Precedence of Operations	3-9
3.4.2 Arithmetic Operators	3-10
3.4.2.1 Integer Division and Modulus Arithmetic	3-12
3.4.2.2 Overflow and Division by Zero	3-13
3.4.3 Relational Operators	3-13
3.4.4 Logical Operators	3-14
3.4.5 String Operators	3-17
3.5 Type Conversion	3-18
3.6 Functions	3-19
3.6.1 Intrinsic Functions	3-19
3.6.2 User-Defined Functions	3-19

Contents

	Page
3.7 The Keyboard	3-20
3.7.1 Function Keys	3-21
3.7.2 Typewriter Keyboard	3-21
3.7.3 Numeric Keyboard	3-22

Chapter 4 Writing Programs Using the SLE GW-BASIC Editor

4.1 EDIT Command	4-1
4.2 Full Screen Editor	4-1
4.2.1 Writing Programs	4-1
4.2.2 Editing Programs	4-3
4.2.3 Logical Line Definition with INPUT	4-10
4.2.4 Editing Lines with Syntax Errors	4-10

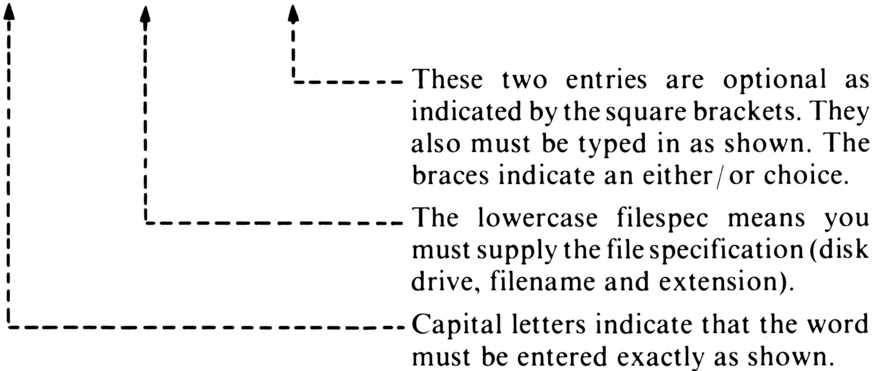
Chapter 5 Working with Files and Devices

5.1 Default Device	5-1
5.2 Device-Independent Input/Output	5-1
5.3 Filenames and Paths	5-2
5.3.1 Filename Specifications	5-2
5.3.2 Pathnames	5-2
5.3.3 Working with Pathnames in BASIC	5-4
5.4 Re-direction of Standard Input and Standard Output	5-5
5.5 Handling Files	5-6
5.5.1 Program File Commands	5-7
5.5.2 Protecting Program Files	5-8
5.6 Data Files: Sequential and Random Access I/O	5-9
5.6.1 Sequential Files	5-9
5.6.1.1 Creating a Sequential File	5-10
5.6.1.2 Reading Data from a Sequential File	5-11
5.6.1.3 Adding Data to a Sequential File	5-12
5.6.2 Random Access Files	5-12
5.6.2.1 Creating a Random Access File	5-13
5.6.2.2 Accessing a Random Access File	5-14

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

Examples

Command Line Explanation



1.3 RESOURCES FOR LEARNING BASIC

This manual provides complete instructions for using Microsoft BASIC. However, no training material for BASIC programming has been provided. If you are new to BASIC or need help in learning programming, we suggest you read one of the following:

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

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

Billings, Karen and Moursund, David. *Are You Computer Literate?* Beaverton, Oregon: Dilithium Press, 1979.

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

Microsoft is a registered trademark and MS and GW are trademarks of Microsoft Corporation.

/F:<number of files>

This switch is ignored unless the /I switch is specified on the command line. Please refer to the /I switch documentation below.

If this switch and the /I switch are present, the maximum number of files that may be open simultaneously during the execution of a BASIC program is set to <number of files>. Each file requires 62 bytes for the File Control Block (FCB) plus 128 bytes for the data buffer. The data buffer size may be altered via the /S: option switch. If the /F: option is omitted, the number of files is set to 3.

The number of open files that SLE MS-DOS supports depends upon the value of the FILES= parameter in the CONFIG.SYS file. It is recommended that FILES=10 for BASIC. Keep in mind that the first 3 are taken by Stdin, Stdout, Stderr, Stdaux, and Stdprn. One additional handle is needed by BASIC for LOAD, SAVE, CHAIN, NAME, and MERGE. This leaves 6 for BASIC File I/O, thus /F:6 is the maximum supported by SLE MS-DOS when FILES=10 appears in the CONFIG.SYS file.

Attempting to OPEN a file after all the file handles have been exhausted will result in a "Too many files" error.

/S:<lrecl>

This switch is ignored unless the /I switch is specified on the command line. Please refer to the /I switch documentation below.

If this switch and the /I switch are present, then the maximum record size allowed for use with random files is set to <lrecl>. NOTE: the record size option to the OPEN statement cannot exceed this value. If the /S: option is omitted, the record size defaults to 128 bytes.

/C:<buffer size>

If present, this switch controls RS232 Communications. If RS232 cards are present, /C:0 disables RS232 support. Any subsequent I/O attempts will result in a "Device Unavailable" error. Specifying /C:<n> allocates space for communications buffers. The amount of space allocated is dependent on the machine-specific portion of SLE GW-BASIC.

/D

If present, this switch causes the Double Precision Transcendental math package to remain resident. If omitted, this package is discarded and the space is freed for program use.

/I

SLE GW-BASIC is able to dynamically allocate space required to support file operations. For this reason, SLE GW-GASIC does not need to support the /S and /F switches. However, certain applications have been written in such a manner that certain BASIC internal data structures must be static. In order to provide compatibility with these BASIC programs, SLE GW-BASIC will statically allocate space required for file operations based on the /S and /F switches when the /I switch is specified.

/M:[<highest memory location>] [,<max block size>]

When present, this switch sets the highest memory location that will be used by BASIC. BASIC will attempt to allocate 64K of memory for the data and stack segment. If machine language subroutines are to be used with BASIC programs, use the /M: switch to set the highest location that BASIC can use. When omitted or 0, BASIC attempts to allocate all it can up to a maximum of 65536 bytes.

If you intend to load things above the highest location that BASIC can use, then use the optional parameter <maximum block size> to preserve space for them. This is necessary if you intend to use the SHELL statement (see Section 7.131). Failure to do so will result in COMMAND being loaded on top of your routines when a SHELL statement is executed.

<maximum block size> must be in paragraphs (byte multiples of 16). When omitted, &H1000 (4096) is assumed. This allocates 65536 bytes (65536= 4096 × 16) for BASIC's data and stack segment. For example, if you wanted 65536 bytes for BASIC and 256 bytes for machine language subroutines, then use /M:,&H1010 (4096 paragraphs for BASIC + 16 paragraphs for your routines).

This option can also be used to shrink the BASIC block in order to free more memory for shelling other programs. /M:,2048 allocates 32768 bytes for data and stack. /M:32000,2048 allocates 32768 bytes maximum, but BASIC will only use the lower 32000. This leaves 768 bytes for the user.

NOTE

<number of files>, <lrecl>, <buffer size>, <highest memory location>, and <maximum block size> are numbers that may be decimal, octal (preceded by &O), or hexadecimal (preceded by &H).

Example:

A>BASICA PAYROLL	Use 64K of memory and 3 files, load and execute PAYROLL.BAS.
A>BASICA INVENT/I/F:6	Use 64K of memory and 6 files, load and execute INVENT.BAS.
A>BASICA /C:0/M:32768	Disable RS232 support and use only the first 32K of memory. The memory above that is free for the user.
A>BASICA /I/F:4/S:512	Use 4 files and allow a maximum record length of 512 bytes.
A>BASICA TTY/C:512	Use 64K of memory and 3 files. Allocate 512 bytes to RS232 receive buffers and 128 bytes to transmit buffers, load and execute TTY.BAS.

2.3 MODES OF OPERATION

The SLE GW-BASIC Interpreter may be used in either of two modes: direct mode or indirect mode.

In direct mode, statements and commands are executed as they are entered. They are not preceded by line numbers. After each direct statement followed by a carriage return, the screen will display the "Ok" prompt. 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 the SLE 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 later be stored in memory. The program stored in memory is executed by entering the RUN command.

2.4 LINE FORMAT

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

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

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

An SLE 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.

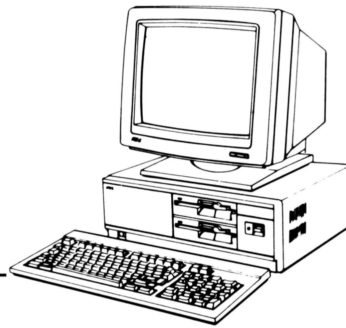
A line may contain a maximum of 255 characters.

With the interpreter, you can extend a logical line over more than one physical line by entering a <linefeed>. <linefeed> lets you continue typing a logical line on the next physical line without entering a <carriage return>. Alternatively, you may type up to 255 characters on a logical line without issuing either a line feed or a carriage return; the text is wrapped and continues on the next physical line.

A period (.) may be used in EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

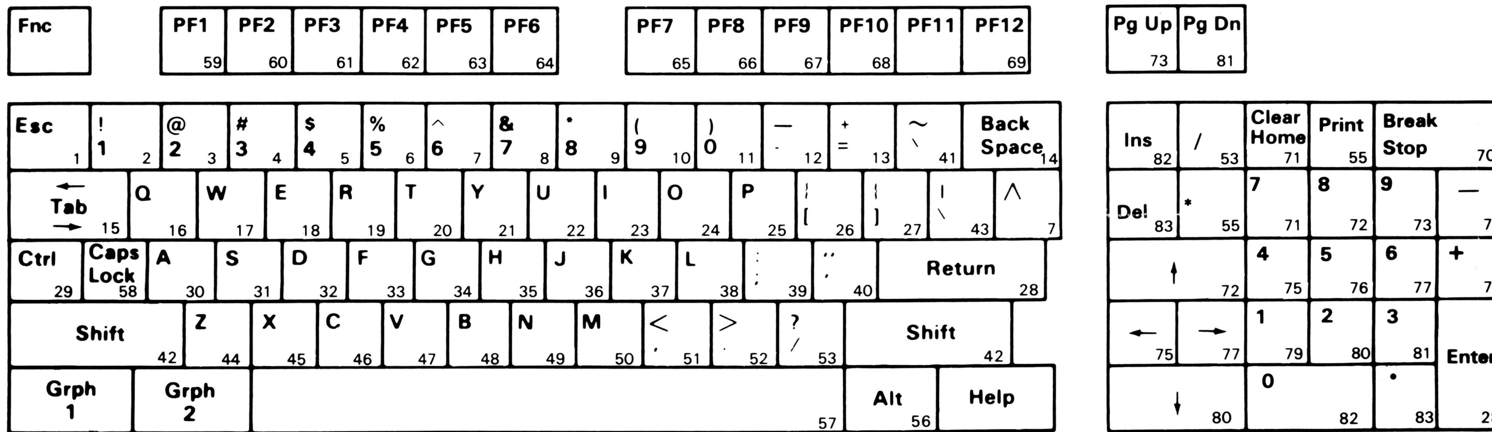
2.5 ACTIVE AND VISUAL (DISPLAY) PAGES

The size of these pages is set by the SCREEN statement. (See SCREEN Statement, Section 7.128.)



Appendix E

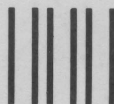
Keyboard Diagram and Scan Codes



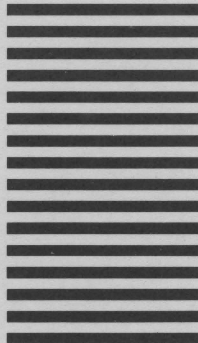
NOTE: THE NUMBER AT THE LOWER RIGHT CORNER OF THE KEY INDICATES THE KEY POSITION.

KEY POSITION	SCAN CODE IN HEX	KEY POSITION	SCAN CODE IN HEX	KEY POSITION	SCAN CODE IN HEX	KEY POSITION	SCAN CODE IN HEX	KEY POSITION	SCAN CODE IN HEX	KEY POSITION	SCAN CODE IN HEX
1	01	15	0F	29	1D	44	2C	58	3A	72	48
2	02	16	10	30	1E	45	2D	59	3B	73	49
3	03	17	11	31	1F	46	2E	60	3C	74	4A
4	04	18	12	32	20	47	2F	61	3D	75	4B
5	05	19	13	33	21	48	30	62	3E	76	4C
6	06	20	14	34	22	49	31	63	3F	77	4D
7	07	21	15	35	23	50	32	64	40	78	4E
8	08	22	16	36	24	51	33	65	41	79	4F
9	09	23	17	37	25	52	34	66	42	80	50
10	0A	24	18	38	26	53	35	67	43	81	51
11	0B	25	19	39	24	54	36	68	44	82	52
12	0C	26	1A	40	25	55	37	69	45	83	53
13	0D	27	1B	41	26	56	38	70	46		
14	0E	28	1C	43	2B	57	39	71	47		

REMIT ONLY
THIS GRAY CARD
TO REGISTER THIS SOFTWARE
PRODUCT. IGNORE ANY
ADDITIONAL CARDS YOU MAY
FIND WITHIN THIS
PACKAGE.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 105 BOXBOROUGH, MA

POSTAGE WILL BE PAID BY ADDRESSEE

NEC Information Systems, Inc.
1414 Massachusetts Ave.
Boxborough, MA 01719

Attention: Customer Technical Services

NEC INFORMATION SYSTEMS PERSONAL COMPUTER
LIMITED WARRANTY
SOFTWARE REGISTRATION CARD

PLEASE READ THE FOLLOWING TEXT CAREFULLY. IT CONSTITUTES A CONTINUATION OF THE PROGRAM LICENSE AGREEMENT FOR THE SOFTWARE APPLICATION PROGRAM CONTAINED IN THIS PACKAGE.

If you agree to all the terms and conditions contained in both parts of the Program License Agreement, please fill out the detachable postcard and return it to:

NEC Information Systems, Inc.
1414 Massachusetts Ave.
Boxborough, MA 01719

Attention: Customer Technical Services

LIABILITY

In no event shall the copyright holder, the original licensor nor any intermediate sublicensors of this software be responsible for any indirect or consequential damages or lost profits arising from the use of this software.

COPYRIGHT

The name of the copyright holder of this software must be recorded exactly as it appears on the label of the original diskette as supplied by NECIS on a label attached to each additional copy you make.

You must maintain a record of the number and location of each copy of this program.

All NECIS software programs and copies remain the property of the copyright holder, though the physical medium on which they exist is the property of the licensee.

MERGING, ALTERATION

Should this program be merged with or incorporated into another program, or altered in any way by the licensee, the terms of the Warranty contained herein are voided and neither NECIS nor the copyright holder nor any intermediate sublicensors will assure the conformity of this software to its specification nor refund the license fee for such nonconformity.

Upon termination of this license for any reason, any such merged or incorporated programs must be separated from the programs with which they have been merged or incorporated and any altered programs must be destroyed.

Proper completion of the card below requires the following:

- Print or type all information.
- Submit one card per software product, regardless of the number of binders and diskettes comprising the package.
- In a package with more than one diskette, use information from the FIRST PROGRAM MASTER diskette for the product serial number.
- Registration cards will not be processed if the hardware serial number from the bottom of your APC III system unit is not included.

"The undersigned Enduser of this NECIS software hereby acknowledges that he/she has read and fully understands the terms and conditions of this license agreement, hereby incorporated into this card and acknowledged by this reference.

The undersigned acknowledges that by signing this document, he/she becomes a party to said license with NECIS, and agrees to be bound by all terms, conditions, and obligations contained therein."

PRODUCT NAME ______________________

PRODUCT SERIAL NUMBER ______________________

PURCHASER INFORMATION

NAME ______________________
(individual, organization, or company)

ATTENTION TO ______________________

ADDRESS ______________________

CITY ____________ STATE OR PROVINCE ____________ ZIP ______

COUNTRY ____________ TELEPHONE (   ) ____________

APC III SERIAL NUMBER ______________________
(you will find this number on the bottom of your SYSTEM unit)

DEALER INFORMATION

DEALERSHIP NAME ______________________

ADDRESS ______________________

CITY ____________ STATE ____________ ZIP ______

DATE PURCHASED ______________________

819-150237-000 Rev. 00