BASIC

Reference Guide

Sr.Partner™
Portable Computer

Panasonic

SOFTWARE LICENSE AGREEMENT

THE SOFTWARE PROGRAM PROVIDED WITH THIS DOCUMENT IS FURNISHED UNDER A LICENSE AND MAY BE USED ONLY IN ACCORDANCE WITH THE LICENSE TERMS DESCRIBED BELOW. USE OF DISK OR THE ACCOMPANYING MANUAL SHALL BE DEEMED TO CONSTITUTE YOUR ACCEPTANCE OF THE TERMS OF THIS LICENSE.

Panasonic Industrial Company, Division of Matsushita Electric Corporation of America ("PIC") provides this Program and licenses its use in the United States and Canada under the following terms and conditions:

1. You may use the Program only on the single Panasonic Sr. Partner computer with which the Program was provided;
2. You may copy the Program into any machine readable or printed form for backup or modification purposes in support of your use of the Program on the single Panasonic Sr. Partner Computer;
3. You may transfer the Program and license it to another party if the other party agrees to accept the terms and conditions of this Agreement. At the time of such a transfer you must also transfer all copies, whether in printed or machine readable form, to the same party or destroy any copies not so transferred;
4. You may not remove any copyright, trademark or other notice or product identification from the Program and you must reproduce and include any such notice or product identification on any copy of the Program.

The Program contains unpublished materials, and the existence of any copyright notice shall not mean that publication has occurred or that all or any part of the Program is not secret.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PROGRAM, OR ANY COPY OF THE PROGRAM, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION OR MERGED PORTION OF THE PROGRAM TO ANOTHER PART, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

This license is effective until terminated. You may terminate it at any time by destroying the Program, together with all copies in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. Upon such termination you must destory the Program together with all copies, modifications and merged portions in any form.

LIMITED WARRANTY
MEDIUM ON WHICH SOFTWARE PROGRAM FOR SR. PARTNER IS RECORDED

This limited warranty applies only to the medium on which the Software Program is recorded. Except for this limited warranty on the medium, Panasonic Industrial Company makes no warranties, express or implied, with respect to the Software Program, its medium, the user manual or the results, use or performance.

If, as a result of faulty manufacture, a defect occurs in the medium on which the Software Program is recorded, and User returns it postage prepaid to anthorized Panasonic & Service Dealer or Matsushita Engineering & Service Company, Industrial Service Division, One Panasonic Way, Secaucus NJ 07094 within sixty (60) days from licensing by User, accompanied by proof of licensing and an explanation of the suspected defect, Panasonic Industrial Company will, at its option, replace the medium free or charge or return of credit an appropriate portion of the license fee paid by User.

This limited warranty applies only to the initial User and does not apply if the product has been subjected to physical abuse or used in defective or non-compatible equipment.

THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE FOR A PARTICULAR PURPOSE, IF ANY, AS TO THE MEDIUM ON WHICH THE SOFTWARE PROGRAM IS RECORDED ARE LIMITED TO SIXTY (60) DAYS FROM THE DATE OF LICENSING BY THE INITIAL USER OF THE PRODUCT AND ARE NOT EXTENDED TO ANY OTHER PARTY. ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE FOR A PARTICULAR PURPOSE WITH RESPECT TO THE SOFTWARE PROGRAM AND MANUAL ARE EXPRESSLY DISCLAIMED.
User agrees that any liability of Panasonic Industrial Company hereunder, regardless of the form of action, shall not exceed the license fee paid by user to Panasonic Industrial Company.

Panasonic Industrial Company shall not be liable for incidental or consequential damages, such as, but not limited to, loss or injury to business, profits, goodwill, or for exemplary damages, even if Panasonic Industrial Company has been advised of the possibility of such damages.

The remedies stated herein are your sole and exclusive remedies; however, some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitations or exclusions may not apply to you.

To locate an Authorized Servicenter in Your Area within the Continental U.S.A.

DIAL TOLL FREE: 800-447-4700

24 Hours a Day, 7 Days a Week

Requests for assistance in obtaining repairs or technical information...contact any one of the following Panasonic Factory Servicenters

Panasonic Factory Servicenter

Eastern	**Midwest**	**Western**	**Southern**
45 Hartz Way Secaucus, NJ 07094 201-348-7466	425A EAST Algonquin Road Arlington Heights, IL 60005 312-981-4841	6550 Katella Avenue Cypress, CA 90630 714-895-7450	3 Meca Way Norcross, GA 30093 404-925-6855 1825 Walnut Hill Lane Irving, TX 75062 214-256-1387

Correspondence requesting products information should be sent to: Computer Dept., Panasonic Industrial Company, Division of Matsushita Electric Corp of America. 1 Panasonic Way, Secaucus, N.J. 07094

BASIC

Reference Guide

Sr. Partner™
Portable Computer

Panasonic

©Copyright Matsushita Electric Industrial Co., Ltd. 1983

©Copyright Microsoft Corporation 1982, 1983

Sr. Partner™ is a Trademark of Matsushita Electric Industrial Co., Ltd.

USA

Panasonic Industrial Company
Division of Matsushita Electric Corporation of America
One Panasonic Way,
Secaucus, New Jersey 07094

Panasonic Hawaii Inc.
91-238 Kauhi St. Ewa Beach
P.O. Box 774
Honolulu, Hawaii 96808-0774

Panasonic Sales Company
Division of Matsushita Electric of Puerto Rico, Inc.
Ave. 65 De Infanteria, KM 9.7
Victoria Industrial Park
Carolina, Puerto Rico 00630

CANADA

Matsushita Electric of Canada Limited
5770 Ambler Drive, Mississauga,
Ontario L4W 2T3

OTHERS

Matsushita Electric Trading Co., Ltd.
32nd floor, World Trade Center Bldg.,
No. 4-1, Hamamatsu-Cho 2-Chome,
Minato-Ku, Tokyo 105, Japan
Tokyo Branch P.O. Box 18 Trade Center

CONTENTS

CHAPTER 1

INTRODUCTION TO THIS BOOK 1-1

CHAPTER 2

INTRODUCTION TO BASIC 2-1

BASIC AND BASICA 2-2

SUMMARY OF CHANGES 2-3

ACCESSING BASIC 2-7

REDIRECTION OF STANDARD INPUT AND
OUTPUT 2-13

OPERATION MODES 2-15

LINE FORMAT 2-16

BASIC COMPONENTS 2-18
Keywords 2-18
Commands 2-18
Statements 2-18
Functions 2-19
Variables 2-19

CHARACTER SET 2-20

CHAPTER 3

THE KEYBOARD

THE KEYBOARD 3-1

INTRODUCTION 3-2

THE FUNCTION KEYS 3-3

THE TYPEWRITER KEYBOARD 3-4
Special Keys 3-4

THE NUMBER PAD 3-7

CHAPTER 4

USING THE BASIC PROGRAM EDITOR

USING THE BASIC PROGRAM EDITOR 4-1

WHAT IS AN EDITOR? 4-2

HOW THE EDITOR WORKES 4-3

USING THE EDITOR 4-4
The Cursor Control Keys 4-4
The Editing Keys 4-7

CREATING AND EDITING A BASIC PROGRAM 4-9
Adding Lines to the Program in Memory 4-9
Changing the Program in Memory 4-10

HOW TO RUN A BASIC PROGRAM 4-11
Syntax Errors 4-11

CHAPTER 5

DISK FILE PROCEDURES 5-1

INTRODUCTION 5-2
Root Directory 5-3
Current Directory 5-3

NAMING FILES 5-4
Pathnames 5-5

PROGRAM FILE MANAGEMENT 5-6

DATA FILES 5-8
File Functions 5-8
Creating and Accessing Sequential Files 5-9
Example Programs for Sequential Files 5-11
Creating and Accessing Random Access Files 5-13
Example Programs for Random Access Files 5-16

CHAPTER 6

BASIC PROGRAMMING ELEMENTS 6-1

RESERVED WORDS 6-3

NUMERIC REPRESENTATION IN BASIC 6-4
Numeric Precision 6-5
Conversion to a Different Precision 6-5
Formatting Numeric Output 6-6
Formatting Double-Precision Numeric Output ..6-7

CONSTANTS 6-9

VARIABLES 6-10
Naming Variables 6-11
Declaring Variable Types 6-11
Array Variables 6-12

PROGRAM EDITOR

DISK FILE

EXPRESSIONS, OPERATORS AND FUNCTIONS 6-14
Arithmetic Operators 6-15
Relational Operators 6-16
Numeric Comparisons 6-17
String Comparisons 6-17
Logical Operators 6-19
Execution of Numeric Operations 6-22
String Operators 6-23

I/O OPTIONS 6-24
Device Name 6-24
Screen Uses 6-25
Text and Graphics Modes 6-26
Specifying Graphics Coordinates 6-27
Window and View 6-27
Additional I/O Features 6-28

CHAPTER 7

COMMANDS, FUNCTIONS, STATEMENTS, AND VARIABLES 7-1

ABS Function 7-4
ASC Function 7-5
ATN Function 7-6
AUTO Command 7-7
BEEP Statement 7-9
BLOAD Command 7-10
BSAVE Command 7-12

PROGRAMMING ELEMENTS

COMMANDS STATEMENS

CALL Statement 7-14
CDBL Function 7-15
CHAIN Statement 7-16
CHDIR Statement 7-19
CHR$ Function 7-21
CINT Function 7-22
CIRCLE Statement 7-23
CLEAR Command 7-26
CLOSE Statement 7-29
CLS Statement 7-31
COLOR Statement (In Text Mode) 7-32
COLOR Statement (In Graphics Mode) 7-34
COM(n) Statement 7-36
COMMON Statement 7-37
CONT Command 7-38
COS Function 7-40
CSNG Function 7-41
CSRLIN Variable 7-42
CVI, CVS, CVD Function 7-43
DATA Statement 7-44
DATE$ Statement and Variable 7-46
DEF FN Statement 7-48
DEF SEG Statement 7-50
DEF type Statement 7-52
DEF USR Statement 7-54
DELETE Command 7-55
DIM Statement 7-56
DRAW Statement 7-58
EDIT Command 7-65
END Statement 7-66

COMMANDS STATEMENS

ENVIRON Statement 7-67
ENVIRON$ Function 7-69
EOF Function 7-70
ERASE Statement 7-71
ERDEV and ERDEV$ Variables 7-72
ERR and ERL Variables 7-73
ERROR Statement 7-75
EXP Function 7-76
FIELD Statement 7-77
FILES Command 7-79
FIX Function 7-81
FOR and NEXT Statements 7-82
FRE Function 7-85
GET Statement (Files) 7-86
GET Statement (Graphics) 7-87
GOSUB and RETURN Statement 7-89
GOTO Statement 7-91
HEX$ Function 7-92
IF Statement 7-93
INKEY$ Variable 7-96
INP Function 7-98
INPUT Statement 7-99
INPUT# Statement 7-101
INPUT$ Function 7-103
INSTR Function 7-105
INT Function 7-106
KEY Statement 7-107
KEY(n) Statement 7-111
KILL Command 7-112
LEFT$ Function 7-113
LEN Function 7-114

LET Statement 7-115
LINE Statement 7-116
LINE INPUT Statement 7-119
LINE INPUT# Statement 7-120
LIST Command 7-122
LLIST Command 7-124
LOAD Command 7-125
LOC Function 7-126
LOCATE Statement 7-127
LOF Function 7-129
LOG Function 7-130
LPOS Function 7-131
LPRINT and LPRINT USING Statements 7-132
LSET and RSET Statements 7-134
MERGE Command 7-136
MID$ Function and Statement 7-137
MKDIR Statement 7-139
MKI$, MKS$, MKD$ Functions 7-141
NAME Command 7-142
NEW Command 7-143
OCT$ Function 7-144
ON COM(n) Statement 7-145
ON ERROR Statement 7-147
ON... GOSUB and ON... GOTO Statements 7-149
ON KEY(n) Statement 7-151
ON PLAY(n) Statement 7-154
ON STRIG(n) Statement 7-156
ON TIMER(n) Statement 7-158
OPEN Statement 7-160
OPEN "COM... Statement 7-165
OPTION BASE Statement 7-172
OUT Statement 7-173

COMMANDS STATEMENS

PAINT Statement 7-175
PEEK Function 7-181
PLAY Statement 7-182
PLAY Statement (ON, OFF, STOP) 7-186
PLAY (n) Function 7-187
PMAP Function..7-188
POINT Function...7-190
POKE Statement 7-192
POS Function 7-193
PRINT Statement 7-194
PRINT USING Statement...........................7-197
PRINT# and PRINT# USING Statements 7-203
PSET and PRESET Statement 7-205
PUT Statement (Files) 7-206
PUT Statement (Graphics) 7-207
RANDOMIZE Statement 7-211
READ Statement 7-213
REM Statement 7-215
RENUM Command.....................................7-216
RESET Command 7-218
RESTORE Statement 7-219
RESUME Statement 7-220
RETURN Statement 7-222
RIGHT$ Function 7-223
RMDIR Statement 7-224
RND Function 7-226
RUN Command 7-228
SAVE Command 7-230
SCREEN Function 7-232
SCREEN Statement 7-234
SGN Function 7-237
SHELL Statement 7-238

SIN Function 7-241
SOUND Statement 7-242
SPACE$ Function 7-245
SPC Function 7-246
SQR Function 7-247
STICK Function 7-248
STOP Statement 7-249
STR$ Function 7-251
STRIG Statement and Function 7-252
STRIG(n) Statement 7-254
STRING$ Function...................................7-255
SWAP Statement 7-256
SYSTEM Command 7-257
TAB Function 7-258
TAN Function 7-259
TIMER Function 7-260
TIMER Statement 7-261
TIME$ Variable and Statement 7-262
TRON and TROFF Commands 7-264
USR Function 7-265
VAL Function 7-266
VARPTR Function 7-267
VARPTR$ Function 7-269
VIEW PRINT Statement 7-270
VIEW Statement 7-271
WAIT Statement 7-274
WHILE and WEND Statements 7-276
WIDTH Statement 7-278
WINDOW Statement 7-281
WRITE Statement 7-286
WRITE# Statement 7-287

APPENDIX A. ERROR MESSAGES A-1

APPENDIX B. MATHEMATICAL FUNCTIONS B-1

APPENDIX C. ASCII CHARACTER CODES C-1

APPENDIX D. ACCESSING MACHINE LANGUAGE SUBROUTINES D-1

APPENDIX E. CONVERTING A PROGRAM TO PANASONIC BASIC E-1

APPENDIX F. EXECUTING APPLICATION PROGRAMS F-1

APPENDIX G. COMMUNICATION I/O PROCEDURES G-1

APPENDIX H. EXAMPLE PROGRAMS H-1

APPENDIX I. INDEX I-1

APP. A
APP. B
APP. C
APP. D
APP. E
APP. F
APP. G
APP. H
APP. I

CHAPTER 1

INTRODUCTION TO THIS BOOK

This book was written to assist you in programming your new Panasonic computer, Sr. Partner™, both by making you aware of its potential for many applications, and by helping you to most efficiently realize this potential in ways that will be useful to you.

A program is a set of instructions for achieving a result via a process. Directing a computer—or a person, for that matter—to perform a task is first a matter of communication in a common language.

Your Sr. Partner "speaks" English and understands BASIC, a cryptic but specific language composed of words quite similar to everyday English. Many of these are verbs that symbolize processes and tell the computer what to do. BASIC also has nouns, which represent numbers and words that are manipulated in performing the operations specified by the verbs. There are even adjectives that describe, for one example, the degree of accuracy of numeric nouns. BASIC also contains punctuation which, like punctuation in any language, affects the meaning of the words.

This book is designed to move you into the computer system quickly, acquainting you with the characteristics of BASIC and Sr. Partner as you will encounter them.

Chapter 2, "Introduction to BASIC", explains the fundamentals of communicating with BASIC, and also briefly describes the components of which BASIC is built.

Chapter 3, "The Keyboard", describes Sr. Partner keyboard, your primary input device for entering programs and BASIC commands.

Chapter 4, "Using the BASIC Program Editor", continues the description of the keyboard in terms of the screen editing functions associated with some of the keys. By the time you finish this chapter, you will have learned to use the Program Editor, a very powerful software device for entering and modifying program text, and should be able to enter and run a program.

Chapter 5, "Disk File Procedures", explains how the Sr. Partner's disk storage system works. It describes how programs and information are placed on the disk, and how they are retrieved for modification or use.

Chapter 6, "BASIC Programming Elements", goes into greater detail than the first five chapters, on a number of topics touched on earlier, for example: how BASIC deals with numbers, how variables are used, how operations are performed, and how to use BASIC for such extended functions as the programming of color graphic applications.

Chapter 7, "Commands, Functions, Statements, and Variables", goes into greater detail than the entire previous book, and with very good reason: in this chapter every component of BASIC is described in terms of its purpose, use, format, and structure. Each description also gives notes and examples, as appropriate.

Also included are a number of appendices. Some of these are intended as supplemental information for the more experienced and technically-inclined user. Two of them, however, should be of immediate assistance to the new programmer: Appendix A, "Error Messages", which lists and explains all the Panasonic BASIC Error Messages; and Appendix H, "Example Programs", which shows you how you can use BASIC to solve a problem and provides a context for some of the BASIC elements you will learn in other parts of the book. Appendix F, "Executing Application Programs" shows the procedure to recover from a system error during executing an application program.

MEMO

CHAPTER 2

INTRODUCTION TO BASIC

BASIC AND BASICA 2-2

SUMMARY OF CHANGES 2-3

ACCESSING BASIC 2-7

REDIRECTION OF STANDARD INPUT AND
OUTPUT 2-13

OPERATION MODES 2-15

LINE FORMAT 2-16

BASIC COMPONENTS 2-18

Keywords 2-18
Commands 2-18
Statements 2-18
Functions 2-19
Variables 2-19

CHARACTER SET 2-20

BASIC AND BASICA

We offer two kinds of BASIC, named BASIC and BASICA which means Advanced BASIC. Advanced BASIC contains the additional statements and functions as follows:

CIRCLE Statement
COM(n) Statement
DRAW Statement
GET Statement (Graphic)
KEY(n) Statement
LINE Statement with *styling*...BASIC 2.0 only
ON COM(n) Statement
ON KEY(n) Statement
ON PLAY(n) Statement...BASIC 2.0 only
ON STRIG(n) Statement
ON TIMER(n) Statement...BASIC 2.0 only
PAINT Statement
PLAY Statement
PLAY (ON, OFF, STOP) Statement...BASIC 2.0 only
PLAY(n) Function...BASIC 2.0 only
PMAP Function...BASIC 2.0 only
POINT(n) Function...BASIC 2.0 only
PUT Statement (Graphic)
RETURN Statement with *line #*
STRIG(n) Statement
TIMER Statement...BASIC 2.0 only
VIEW Statement...BASIC 2.0 only
VIEW PRINT Statement...BASIC 2.0 only
WINDOW Statement...BASIC 2.0 only

When you make your own program, we recommend you use BASICA, because BASIC is a subset of BASICA. In case the paticular programs can only run under BASIC, you have to follow the instructions of the programs.

We use the word "BASIC" in this manual as the general name of BASIC and BASICA described above.

SUMMARY OF CHANGES

BASIC 2.0 has expanded the options allowable in the command line:

The /**M** switch has been enhanced to include the new optional parameter maximum block size which allows you to load programs above the BASIC workspace.

The optional /**D** switch allows calculation in double precision. You can use this option with the ATN, COS, EXP, LOG, SIN, SOR, and TAN functions.

Two new specifications <stdin and >stdout, allow redirection of standard input and standard output.

BASIC 2.0 now uses tree-structured directories to improve file organization. This is especially helpful when working with a hard disk. Three new commands provide directory management support:

MKDIR makes a new directory on the disk.

CHDIR changes the current directory.

RMDIR removes a directory from the disk.

In addition, filespec has been expanded to allow a "directory path" to be specified in the command or statement syntax. Commands which allow **path** are: BLOAD, BSAVE, FILES, KILL, LOAD, MERGE, NAME, RUN, and SAVE. The OPEN and CHAIN statements can also include the **path.**

BASIC 2.0 contains improved disk I/O facilities for larger files:

Enhancements to the **GET** and **PUT** statements now allow record numbers in the range of 1 to 16,777,215. This enables BASIC to accommodate large files with short record lengths.

LOC now returns the actual record position within the file for random files. For sequential files **LOC** returns the current byte position divided by 128.

For both random and sequential files, **LOF** returns the size of the file in number of bytes.

Graphics statements and functions have been improved.

Full line clipping. Points plotted outside of the screen or viewport do not appear and no longer wrap around. Lines that intersect the screen or viewport will appear, cross the screen and disappear at the other end. Line clipping is used by the **CIRCLE, DRAW, LINE, POINT, PSET, PRESET, PAINT, VIEW and WINDOW** statements.

A new option, style, is now supported by the **LINE** statement. Using hexadecimal values, style plots a pattern of points on the screen.

Tiling is now allowed because of the addition of the **background** option to the **PAINT** statement.

DRAW has two new options. **TA(n)** rotates angle n from −360 to 360 degrees. **P paint, boundary** sets paint and border attributes.

The **POINT** function can now be expressed in the form **v=POINT(n).** This returns the current value of the **x** or **y** coordinate.

The new **VIEW** statement is used to define a viewport (or area) within the screen. The boundaries selected must be within the physical limits of the screen.

The new **WINDOW** statement allows redefinition of screen coordinates.

The new **PMAP** function allows BASIC to map expressions to logical or physical coordinates.

BASIC 2.0 has added additional event trapping:

ON KEY(n), KEY(n) and **KEY** statements have been expanded to allow six additional user defined keys.

The new **ON PLAY(n)** statement allows continuous music to play during program execution.

The new **ON TIMER(n)** statement transfers control to a specified line number when the defined time period has elapsed.

Additional enhancements:

EOF(0) returns the end of file condition on redirected standard input devices.

Two new options are available for the **PLAY** statement. **>n** increments the note one octave. **<n** decrements the note one octave.

DELETE now allows line deletions from a given line number to the end of the program.

RANDOMIZE no longer forces floating point values to integer (allows single- or double-precision expressions). Using TIMER allows the generation of a new random number without a prompt.

OPEN "COM... has a new option, **PE** which allows for parity checking.

Pressing **<Ctrl> <PrtSc>** prints all text that appears on the screen on your system printer. This differs from **<Shift> <PrtSc>** in that all text which appears on the screen is printed until you press **<Ctrl> <PrtSc>** again.

Additional new features:

Two Read only System variables, **ERDEV** and **ERDEV$,** allow for more precise error reporting. ERDEV holds the value at the time of device error. ERDEV$ holds the device name where the error occurred.

The **SHELL** statement allows COMMAND or "Child processes" to run without exiting BASIC.

The **VIEW PRINT** statement defines the boundaries of the text window. The screen editor limits functions such as cursor movement and scrolling to the text window.

You can now modify parameters in BASIC's Environment String Table using the **ENVIRON** statement. And the **ENVIRON$** function allows you to retrieve the specified Environment string from BASIC's Environment table.

The **PLAY** function returns the number of notes currently in the background music buffer.

The **TIMER** function returns the number of seconds elapsed since system reset or midnight.

ACCESSING BASIC

BASIC[A] [*filespec*] [<*stdin*] [[>]>*stdout*] [/F:*files*] [/S:*bsize*] [/C:*combuffer*] [/M:[*max workspace*][,*max blocksize*]] [/D] [/I]

Preparing your Sr. Partner for use is a two-step process. First the operating system (calls DOS, for Disk Operating System) must be loaded; then you must inform DOS that your communication will be in the BASIC language.

To load DOS:

1. Turn on the computer.

2. Place the DOS disk in Drive A.

The computer will display the DOS prompt symbol (A>). To start BASIC or BASICA.

1. Enter the word BASIC or BASICA.

The computer will display the BASIC version designation and release number, then the number of available bytes and the BASIC prompt symbol (Ok), indicating that it is ready to receive commands from you.

Later, when you have finished your work in BASIC and are ready to return to DOS, exit BASIC by entering:

```
SYSTEM
```

Options can be included when you specify BASIC or BASICA command. These options are used to specify the size of a storage which BASIC uses to hold programs and data and for buffer areas. You can also immediately load and run a program by these options.

Command options are defined as follows:

filespec — represents the file specification of a program to be immediately loaded and run. It must be a string constant (but *not* enclosed in quotes). If the filename is eight characters long or less, and no extension is given, .BAS is supplied as the default. If *filespec* is included, BASIC proceeds as though RUN *filespec* was the first thing entered after it was ready. Also, when you specify *filespec*, the BASIC screen with the copyright notices is not displayed.

<*stdin* — allows BASIC to receive input from other than the keyboard (standard input device). The <*stdin* option instructs BASIC to receive input from the specified file. Position <*stdin* before any switches. See "Redirection of Standard Input and Standard Out put" for further information.

>*stdout* — allows BASIC to write output to other than the screen (standard output device). The >*stdout* option instructs BASIC to write output to the specified file or device. Position >*stdout* before any switches. See "Redirection of Standard Input and Standard Output" for further information.

Options that start with a slash (/) are called switches, which are used to specify parameters. The following BASIC command options are switches:

/F:*files* — sets the maximum number of files that may be open at any given time while a BASIC program is running.
Each file requires 188 bytes of memory for the file control block, plus the buffer size specified by the /S: switch. If the /F: switch is omitted, the number of files will default to three.

The actual number of files that may be open at any one time depends on the value of the FILES= parameter in the DOS configuration file, CONFIG.SYS, with the default being FILES=8. BASIC uses three files by default, and leaves five files for BASIC file I/O. Thus, /F:5 is the maximum value you can specify when FILES=8 and you want to be able to have all files open at once.

/S:*bsize* Set the random file buffer size. The record length given with the OPEN statement may not exceed this value. The default *bsize* is 128 bytes, and the maximum allowable value is 32767 bytes.

/C:*combuffer* sets the receiving buffer's size when using the internal or optional serial interface, and has not effect unless you are equipped with this adapter. The tranamit buffer for communications is always set to 128 bytes. The maximum value you may enter for this switch is 32767 bytes. If /C: is omitted, the default allocation is 256 bytes. If you have a high-speed line, you should use /C:1024. If you have two serial interfaces on your system, both receive buffers will be set to the size specified by /C:. You may deactivate a serial interface by giving /C: a value of zero, in which case no buffer space will be reserved for communications, and communications support will not be included when BASIC is loaded.

/M:*max workspace*

sets the maximum number of bytes that can be used as BASIC workspace. BASIC can use a maximum of 64k bytes of memory, so the highest value you may set is 64k (hex FFFF). This option may be used to reserve space for machine language subroutines, or to store special data. If /M: is omitted, BASIC uses all available memory up to a maximum of 64k bytes.

,*max blocksize*

reserves space for the workspace and your programs. This parameter must be allocated in Paragraphs (byte multiples of 16). If the parameter is omitted, 4096 (&H1000) is assumed and 65536 bytes, equal to 4096 x 16, are reserved for BASIC's Data and Stack segment. To allocate 65536 bytes for BASIC and 512 bytes for machine language subroutines, use /M: &H1010 (4096 paragraphs for BASIC plus 16 paragraphs for your routines). You must include this option when you utilize the SHELL statement. Otherwise, COMMAND will be loaded on top of your routines when a SHELL statement is executed. Use this option to decrease the BASIC block so that more memory is free for SHELLing other programs. /M:,2048 allocates 32768 bytes for data and stack (2048 x 16). /M:32000,2048 allocates to BASIC 32768 bytes as the maximum, however BASIC will only use the lower 32000. 768 bytes are left for your programs.

/D

tells BASIC to use double-precision transcendental functions. /D uses about 3,000 additional bytes for transcendental functions. The following functions can be converted to double precision: ATN, COS, EXP, LOG, SIN, SQR, and TAN. When /D is omitted this function is discarded, and the space is freed for other program use.

/I

Memory for file operations will be dynamically allocated by BASIC. When the /I switch is used the /S and /F switches are ineffective. When the /I switch is not used BASIC will allocate memory according to the /S and /F switches i.e. static allocation.
When memory is dynamically allocated for file operations the area where strings are stored will move accordingly.

WARMING: When loading machine language programs into the string area the programs may move or be destroyed.

NOTE: The options *files, max workspace, bsize,* and *combuffer,* and *max blacksize* are all entered as numbers that may be either decimal, octal (preceded by &O), or hexadecimal (preceded by &H) representations.

Some examples are following:

The program LISTING.BAS is loaded and run with all defaults:

```
BASIC LISTING.BAS
```

Here BASIC is started with to use all memory and four files, and PROG.BAS loaded and run:

```
BASIC PROG/F:4
```

This command starts BASIC so that the maximum workspace size is 32K bytes of memory. All other defaults apply.

```
BASIC/M:32768
```


This command sets the maximum workspace size as HEX 9000, which means up to 36K bytes of memory. Also, a file control block is set up for one file, and the program PROG2.TST is loaded and run.

```
BASICA PROG2.TST/F:1/M:&H9000
```

This command starts BASIC and instructs that all transcendental functions be calculated in double precision.

```
BASIC /D
```

REDIRECTION OF STANDARD INPUT AND OUTPUT

BASIC 2.0 allows you to redirect your BASIC input and output. Standard input is usually read from the keyboard. Now, it can now be redirected to be read from any file you specify on the command line. Standard output is usually written to the screen. Now, it can be written to any file or device you specify on the command line.

BASIC filespec [<stdin] [[>]>stdout]

Some examples are following:

Data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the keyboard as usual. Data written by PRINT will go into the NEW.OUT file

```
BASIC STAT >NEW.OUT
```

Data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from REVISED.DAT file. Data written by PRINT will go to the screen as usual.

```
BASIC STAT <REVISED.DAT
```

Data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from JUNE.DAT file. Data written by PRINT will go into the JULY.DAT file.

```
BASIC STAT <JUNE.DAT >JULY.DAT
```

Data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the\INVEST\TAX\IRA file. Data written by PRINT will be added to the\INVEST\TAX.DAT file.

```
BASIC STAT <\INVEST\TAX\IRA >> \INVEST \TAX.DAT
```

INTRO TO BASIC

Notes on using redirection of standard input and output:

Redirection of standard input causes all INPUT, INPUT$, INKEY$, and LINE INPUT statement to read from the specified input file. Input will not be read from the keyboard.

Redirection of standard output causes all PRINT statements to write to the specified output file or device.

Error messages will still go to redirected standard output and to the screen. Error messages cause all files to be closed, the program to end, and control to return to DOS.

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

File output to "SCRN:" still outputs to the screen.

When the ON KEY(n) statement is used, BASIC continues to trap keys from the keyboard.

When standard output has been redirected, <Ctrl> <PrtSc> no longer prints the screen.

EOF(0) returns the end of file condition on redirected standard input devices.

<Ctrl> <Break> returns to standard output. All files are closed and control returns to DOS.

OPERATION MODES

When BASIC prompts you for a command by displaying "Ok", you may respond in either of two modes: *direct* and *indirect*.

In Direct Mode, commands are executed immediately and are not saved in storage. Results, however, may be displayed or stored. Direct Mode is useful both in debugging and for computations that do not require programming.

Indirect Mode is used to enter programs, and is signalled to BASIC by entry of a *line number* at the beginning of each program line. The line number causes BASIC to store the commands sequentially in memory.

LINE FORMAT

BASIC program lines are entered in the following format:

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

nnnnn represents the line number, which may be from one to five digits long. Every BASIC program line must begin with a line number; this number indicates the sequence in which the lines will be both stored in memory and executed when the program is run. The initial number may be zero, and the highest permitted is 65529. When using the LIST, AUTO, DELETE, and EDIT commands, a period may be substituted for the line number to refer to the current line.

Line ranges may be specified in the LIST and DELETE commands with a hyphen, for example:

Command	Meaning
LIST 40–60	List Lines 40 through 60 inclusive.
LIST 40–	List all lines from Line 40 to the program's end.
DELETE –40	Delete all lines from the beginning of the program through Line 40 inclusive.
DELETE 40–	Delete all lines from Line 40 through the end of the program, inclusive.

BASIC statement is either *executable* or *non-executable.* An executable statement is a command that specifies the next step in running a program, for example: GOTO 220. A non-executable statement, such as REM, does not cause a specific action when encountered by BASIC. Format and syntax for all BASIC statements are given in Chapter 7.

The square brackets in the example program line at the beginning of this section indicate optional entries and do not appear in a normal line. More than one BASIC statement may be included in a single line, provided each statement is separated from the preceding statement by colon (:). Each line must be at least one character in length, and may total no more than 255 characters.

A *comment* can be added to the end of a line by the ' (single quote).

BASIC COMPONENTS

BASIC components are *keywords, commands, statements, functions* and *variables*. All BASIC keywords are described and listed in the Chapter 6 section titled "Reserved Words". Each BASIC command, statement, function, and variable is described in detail in Chapter 7.

Keywords

Keywords are words with specific meaning to BASIC. When it encounters a keyword, BASIC interprets the word as an instruction, and immediately executes the action specified by the word.

Commands

When BASIC is invoked and the BASIC prompt "Ok" appears, the system is said to be on the interpreter's *command level*, where a BASIC keyword entered by the user will be interpreted as a *command*. BASIC immediately executes keywords it interprets as commands.

Statements

BASIC *statements* are made up of BASIC keywords, either singly or in groups, usually accompanied by data or variables. When a program composed of BASIC statements is run, the statements are executed in sequence and the data manipulated as directed by the keywords.

Functions

BASIC performs numeric and string *functions*. A numeric function is a mathematical calculation, for example: to determine the sine of a given angle of x radians. Integer and single-precision results are returned by numeric functions unless otherwise specified. String functions operate on strings of characters.

For example, the keyword TIME$ returns the time by the system clock, which is represented by a string of numbers. BASIC functions may be user-defined by use of the DEF FN statement.

Variables

A *variable* is a numeric value that has been given a name composed of alphanumeric characters, or a name to which a computed value is assigned. Variables that are BASIC keywords provide information as a program is executed, for example: ERR specifies the latest error that occurred. Other variables are defined and/or computed as the program runs.

CHARACTER SET

The BASIC character set consists of the characters that make up BASIC commands, statments, functions, and variables, and includes:

Alphabetic characters—letters of the alphabet, both capitals and lowercase

Numeric digits—the numbers zero through nine

The following special characters:

Character	BASIC Usage
	blank
=	equal sign or assignment symbol
+	plus sign or concatenation symbol
–	minus sign
*	asterisk or multiplication sign
/	slash or division sign
\	backslash or integer division symbol
^	caret or exponentiation symbol
(	left parenthesis
)	right parenthesis
%	percent sign or integer type declaration character
#	number (or pound) sign or double-precision type declaration character
$	dollar sign or string type declaration character
!	exclamation point or single-precision type declaration character
&	ampersand
,	comma
.	period or decimal point

'	single quotation mark (apostrophe) or remark delimiter
;	semicolon
:	colon or statement separator
?	question mark (PRINT abbreviation)
<	less than
>	greater than
"	double quotation mark or string delimiter
_	underline

The computer is capable of displaying or printing many other characters with no specific meaning to BASIC. Appendix C (ASCII Character Codes) gives a complete list of the computer's character set.

MEMO

CHAPTER 3
THE KEYBOARD

INTRODUCTION 3-2

THE FUNCTION KEYS.. 3-3

THE TYPEWRITER KEYBOARD 3-4

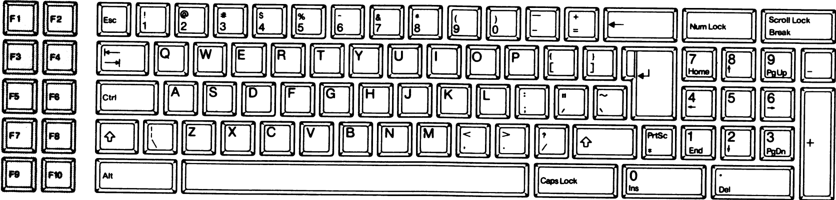
Special Keys 3-4

THE NUMBER PAD 3-7

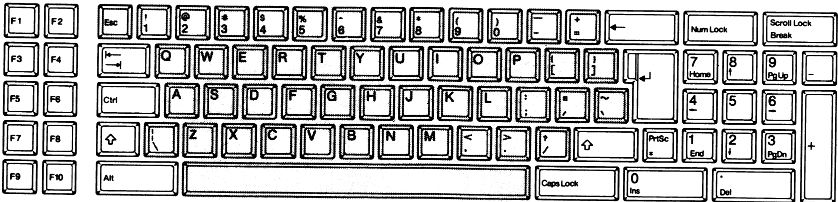
INTRODUCTION

The Sr. Partner's keys are divided into three areas according to their use:

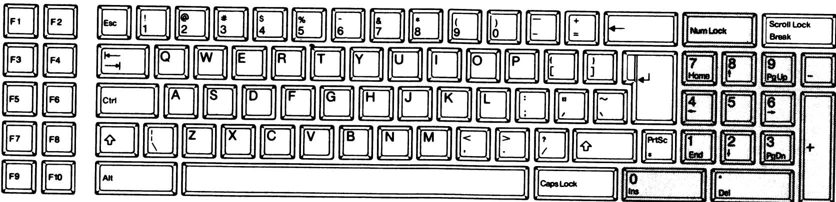
Grouped on the left are the *Function Keys* F1 through F10.



Positioned in the center is the *Typewriter Keyboard*, where are found the normal alphabet and number keys.



On the right is the *Number Pad*, which resembles a calculator keyboard.



Three keys are not discussed in this chapter: ESC on the Typewriter Keyboard, and HOME and END on the Number Pad. Instead, in their context as Program Editor functions, they are explained in Chapter 4 (the BASIC Program Editor).

THE FUNCTION KEYS

In addition to functioning as program interrupts (refer to the ON KEY statement in Chapter 7), the ten Function keys are usually used as what are termed "soft keys". This means that each key can be set so that, when pressed, a designated series of characters will automatically be entered. In fact, the Panasonic comes equipped with some of these keys already set to enter certain frequently-used commands; refer to chapter 7 (the KEY statement) for more information.

THE TYPEWRITER KEYBOARD

The Typewriter Keyboard looks and works very much like a regular typewriter. It contains the letters of the alphabet and the numbers zero through nine, all positioned as usual. It also contains a large variety of special characters. Many of these are familiar (punctuation and such standard symbols as $ and %), though some of them may be new to you. Shifting the Typewriter Keyboard keys is done in the normal fashion, by pressing <SHIFT> key and then pressing another keyboard key.

Special Keys

The long key with the arrow coming down and turning left is the *carriage return*. This key is used to signal an entry to the computer. It is usually referred to as the <ENTER> key. The words "Enter (something)" mean "Type (something) and press the <ENTER> key.

The <Caps Lock> key functions similarly to a Shift Lock. After you press <Caps Lock> and until you press it again, any alphabetic characters you type will appear on the screen in uppercase. However, you will not be able to enter any other shifted characters, such as punctuation and symbols by using the <Caps Lock> key. In Caps Lock mode, lowercase letters may be typed by holding down a Shift Key.

The <⇦> (BACKSPACE) key not only backspace, but also erases as it goes. If you don't want to erase what you have typed, backspace the cursor with the Cursor Left Key (an arrow pointing left on the Number Pad).

The <PrtSc> key, unshifted, will type an asterisk. Shifted, it is used to print the current screen display on the printer.

The <Alt> key has two main functions:

1. If <Alt> is held down while pressing one of the alphabetic keys listed below, the associated BASIC keyword will be typed on the screen:

Letter	Word	Letter	Word
A	AUTO	M	MOTOR
B	BSAVE	N	NEXT
C	COLOR	O	OPEN
D	DELETE	P	PRINT
E	ELSE	R	RUN
F	FOR	S	SCREEN
G	GOTO	T	THEN
H	HEX$	U	USING
I	INPUT	V	VAL
K	KEY	W	WIDTH
L	LOCATE	X	XOR

2. THE <Alt> key is also used to enter characters that are not on the keyboard, but are included in the ASCII Character set. This is done by holding down <Alt>, entering the three-digit ASCII character code, using the Numeric Pad keys.

The <Ctrl> key, in combination with other keys, allows you to enter codes that access certain specialized functions. To enter these key combinations, hold the <Ctrl> key down and press the desired key(s).

The Ctrl Key combinations and the effect of using them are listed below:

<Ctrl> <Break>	This combination stops execution of a program at the next instruction and returns the system to the BASIC command level. It may also be used to exit the AUTO line numbering function.
<Ctrl> <Num Lock>	This key combination causes the computer to pause until you press any key except: the <SHIFT> key, and the <Break> and <Num Lock> key.
<Ctrl> <Alt> <Del>	If the computer is on, this combination will cause a System Reset, which is similar to the effect of turning the computer off and then on. However, the reset is accomplished more quickly when done this way. First press <Ctrl>, then <Alt>, then, holding them down, press <Del>.
<Ctrl> <PrtSc>	This combination of key strokes will cause any text displayed on the computer screen to be sent to the printer. Text appearing on the screen will continue to print until you press both keys again to switch this function off. Note that this operation can slow processing somewhat, the computer will not continue until printing of displayed material is completed.

The Shift Key combination and the effect of using it is listed below:

<Shift> <PrtSc>	If you press and hold <Shift>, then press <PrtSc> and release both, the current screen display will be printed on the printer. To stop printing, press <Ctrl> <Break>.

THE NUMBER PAD

Normally, the keys on the Number Pad are used to manipulate the cursor and to insert or delete characters. These functions are described in detail in Chapter 4 (The BASIC Program Editor).

The Number Pad can be shifted with the <Num Lock> key so that it can be used for calculator functions, the keys generating the numbers zero through nine, plus the decimal point. To return the pad to its cursor control functions, press <Num Lock> again. <Num Lock> can be temporarily reversed by pressing a Shift Key.

The <Scroll Lock>, <Pg Up>, and <Pg Dn> keys are not used in BASIC programming.

MEMO

CHAPTER 4

USING THE BASIC PROGRAM EDITOR

WHAT IS AN EDITOR? 4-2

HOW THE EDITOR WORKS 4-3

USING THE EDITOR 4-4

- The Cursor Control Keys 4-4
- The Editing Keys 4-7

CREATING AND EDITING A BASIC PROGRAM 4-9

- Adding Lines to the Program in Memory 4-9
- Changing the Program in Memory 4-10

HOW TO RUN A BASIC PROGRAM 4-11

- Syntax Errors 4-11

WHAT IS AN EDITOR?

The BASIC Editor is a piece of software used to modify a BASIC program. As directed by the user, the editor will add, delete, or alter specified characters or lines.

The BASIC editor is called a "line editor", which means that it processes changes only to a single program line at one time. Thus, though you can change any line on your screen, and may make as many changes as you like on each line, you must enter the changes one line at a time.

HOW THE EDITOR WORKS

Once you have accessed BASIC and see the BASIC prompt, you are in command mode, where you may enter BASIC commands, and create and edit programs. Any line beginning with a number that you enter will be considered a BASIC program line and will be stored as part of the program in memory.

Note that "storing a program in memory" is not the same as "writing a program file to the disk". Unless specifically deleted via a BASIC command, files written to disk are permanently recorded and accessible as long as the physical disk is intact. The contents of the memory, however, are routinely erased in a number of situations, for example: before entering, editing, or running a program, or even when turning the computer off.

Therefore, if you want to keep what you have entered, be sure to use the SAVE command (see Chapter 5) to name it and write it to the disk when you have finished entering it. If you do not wish to save a program you have entered into memory, you can delete it in its entirety by entering the NEW command (see Chapter 7), which is used to clear the memory.

Similarly, "typing a line" is not the same as "storing it in the program in memory". The editor makes no additions or changes to the program in memory until you press the <ENTER> key.

You should also be aware of the distinction BASIC makes between a physical line and a logical line. A physical line is as long as the horizontal width of the screen. A logical line is a string of text that is processed as a unit, and may extend over more than one physical line. Thus, it is possible to enter the up-to-255 allowable characters in a BASIC program line. The editor remembers where the line started (at the last line number) and knows that it will end at the next line number.

USING THE EDITOR

Your "place" on the screen is "marked" by a blinking underline called the cursor. The position of the cursor indicates where you will start or continue typing, or where insertions, deletions, or alterations will take effect. The cursor's position also indicates the "current line", the line which will be processed by the editor when the <ENTER> key is pressed. (The cursor may be anywhere on the line; the editor knows where the line begins and ends and will process it all.)

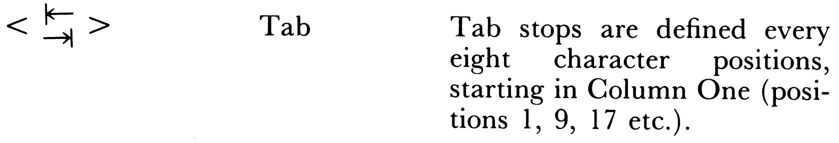
The Cursor Control Keys

The cursor is moved either by typing characters or by pressing one of several keys on the Number Pad called, collectively, Cursor Control Keys. The action of each of these keys, pressed alone or in combination with the <Ctrl> key, is described below.

Key(s)	Function	Action
<Home>	Home	The cursor is moved to the upper left corner of the screen.
<Ctrl> <Home>	Clear Home	The screen is cleared and the cursor moved to the upper left corner.
< ↑ >	Cursor Up	The cursor is moved up one line.
< ↓ >	Cursor down	The cursor is moved down one line.

PROGRAM EDITOR

Key	Function	Description
<←>	Cursor Left	The cursor is moved one position to the left. If the cursor is in column one on the left side of the screen, it is "wrapped" to the last column of the preceding line.
<→>	Cursor Right	The cursor is moved one position to the right. If it is in the last column on the right side of the screen, it appears in the first column of the following line.
<Ctrl> <→>	Next Word	The cursor is moved right to the next word, which is the next alphabetic character or number that is preceded by a blank or special character. For example, in the line that follows, the cursor is positioned under the U in NUM2: `50 (N1,NUM2)-(TOT,75),3,BF` If you press <Ctrl> <→>, the cursor moves to the first character of the next word (TOT). If you enter the combination again, it moves under the 7.

<Ctrl> <←>	Previous Word	The cursor is moved left to the previous word, which is the next letter or number to the left of the cursor that is preceded by a special character or a blank.
<End>	End Line	The cursor is moved to the end of the current line.
	Tab	Tab stops are defined every eight character positions, starting in Column One (positions 1, 9, 17 etc.). When insert mode is off, the Tab Key moves the cursor to the next tab stop. When insert mode is on and Tab is pressed, blanks are inserted from the cursor position to the next tab stop.

The Editing Keys

The editor also includes functions that are specified by a group of Editing Keys. These are used to insert and delete characters, lines, and portions of lines.

Key(s)	Function	Action
<Ins>	Insert Mode	If you are not in Insert Mode, this key puts you there. If Insert Mode is on, pressing <Ins> turns it off. In Insert Mode, characters you type are inserted to the left of the cursor, pushing the cursor and the text that follows it to the right. Lines are "folded"; that is, as characters are pushed off the right side of the screen, they reappear on the left side on the next lines. Insert Mode is automatically turned off when you use any of the Cursor Control Keys or press the <ENTER> key.
<Del>	Delete	The character at the cursor position is deleted. All characters to the right of the cursor are moved one space to the left. Line folding takes place if necessary.
<Ctrl> <End>		All characters from the cursor position to the end of the current line are erased.

<⇦>	Backspace	The character immediately to the left of the cursor is deleted. All characters to the right of the cursor are moved left one position, with line folding occuring if necessary.
<Esc>		The line containing the cursor is erased. This key is only used to delete a line that has not been entered. It does not delete a line from a program that is in memory.
<Ctrl> <Break>		The editor assumes command level, and any changes made to the current line are not entered. The line is not erased; it remains in the state it was in when the last entry was made.

CREATING AND EDITING A BASIC PROGRAM

Adding Lines to the Program in Memory

Creating a BASIC program is a matter of entering program lines into memory. Any BASIC program line:

begins with a line number between 0 and 65529

ends with "Enter", which is specified with the <ENTER> key

contains a maximum of 255 characters and a minimum of one

Before entering program lines, you may edit them as much as you wish, using the cursor control and editing keys. Once entered into the program, any changes you make to them will take effect when you press <ENTER>.

You may enter BASIC keywords and variable names in either upper- or lowercase, or a combination. The editor will convert all input to uppercase except strings enclosed in quotes, DATA statements, and remarks.

You may wish to use the AUTO line-numbering function (see Chapter 7) when entering a program.

When entering or editing a line, you can cause text typed after the cursor position to be placed on the next screen line when you enter the logical line. This is done by pressing <Ctrl> <ENTER>, and is called "typing a line feed". When you press <ENTER> and line is processed, blanks will be inserted from the place where you pressed <Ctrl> <ENTER> to the end of the physical line.

The computer will give an "Out of Memory" error message if you try to add a line to a program that fills the memory.

Changing the Program in Memory

You can display any line or range of lines in the program in memory by using the LIST command (see Chapter 2 and 7). The EDIT command (see Chapter 7) can also be used to display a desired line.

If you enter a line for which the line number already exists, the existing program line will be replaced with the new one.

Selected lines can be deleted by entering the line number only. Or a group of lines may be deleted with the DELETE command (see Chapter 2 and 7).

A line may be duplicated by positioning the cursor on its line number, typing a new number over it, and pressing Enter. Both the line with the original number and a duplicate with the new number will exist. A line number can be changed this way by then deleting the original line.

HOW TO RUN A BASIC PROGRAM

A BASIC program must be in memory to be executed. Transferring a program that you have saved on disk to memory is done with the LOAD command (described in detail in Chapters 5 and 7).

In command mode, respond to the BASIC prompt by entering:

```
load "progname"
```

In this example, "progname" is the name of your program. To execute the program in memory, BASIC displays:

```
Ok
```

Now you may enter:

```
run
```

BASIC will execute the program statements sequentially by line number.

Syntax Errors

If the editor finds a syntax error while running a program, it will display the line in error with the cursor positioned under the first digit of the line number. If you wish, you may edit the line now and the change(s) will be stored in the program. However, when you do this during a program interrupt:

All variables and arrays will be set to zero or null.

Any open files will be closed.

You cannot continue running the program with the CONT command.

If you want to check the contents of a variable before editing the line, press <Ctrl> <Break> and BASIC will return to command level. Since no change was made, the variables will be intact. After examining the variable(s), edit the line in error and rerun the program.

CHAPTER 5

DISK FILE PROCEDURES

INTRODUCTION 5-2
Root Directory 5-3
Current Directory 5-3

NAMING FILES 5-4
Pathnames 5-5

PROGRAM FILE MANAGEMENT 5-6

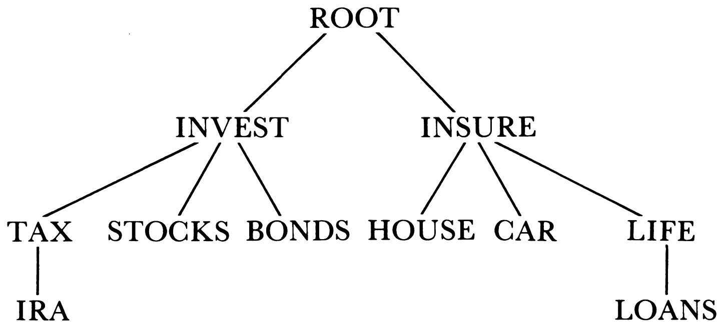
DATA FILES 5-8
File Functions 5-8
Creating and Accessing Sequential Files 5-9
Example Programs for Sequential Files 5-11
Creating and Accessing Random 5-13
Example Programs for Random Access Files ..5-16

INTRODUCTION

The Sr. Partner stores information on disk in sets or related data called *files*. These files are organized into directories that enable you to quickly find the information and programs you need.

BASIC 1.0 used a simple, single-level directory structure that provided good file management capabilities for files stored on floppy disks. Now that BASIC supports hard disks, which can hold hundreds of files, a more advanced directory system is required. This is provided by so-called "stratified", or hierarchical directories. Related files are grouped in different directories on the same disk. Each directory can contain both file names and other directory names.

For example, you could create a hierarchical directory that looks like this:



BASIC provides a way for you to specify a "path" to the file you want.

DISK FILE

Root Directory

When you FORMAT each disk, a single directory is created the *root directory*. In this *root directory* you can store filenames or the names of other directories, known as *sub-directories*. The *sub-directories*, in turn, can hold the names of more files and sub-directories. In the illustration above, the hierarchy can be as deep and wide as disk space permits.

Current Directory

The *current* directory is the default directory for each drive on the Sr. Partner. BASIC will always search the *current* directory for a filename that is specified without a directory name. BASIC will automatically use the root directory as the current directory until you change it with the CHDIR command. (See the CHDIR Command in Chapter 7.)

The procedures described in the sections that follow allow you to create files, access them for input and output purposes, and manipulate them in a number of ways.

A number of BASIC terms and commands are briefly explained in this chapter, which also includes a great many BASIC program statements shown as examples. Both Chapters 6 and 7 provide more detailed information about the concepts presented in this chapter.

NAMING FILES

Each file is referenced by a unique *file specification*, which term is shortened to *filespec* in this book. The filespec may contain a *device*, and must include a *filename*. The format for a BASIC *filespec* is.:

[*device*:]*filename*

If you wish to designate the disk, *device* must be either A: or B:, (if you have the optional drive B). If you wish other devices, refer to "Device Name" in Chapter 6. The colon is required when specifying a device. If you do not specify the disk, BASIC will assume the file is on the currently-loaded disk.

The format for a BASIC *filename* is:

name[.*extension*]

The "name" may be from one to eight characters long. The optional "extension", which requires the delimiting period, may be from one to three characters long. Refer to Chapter 6 for a more detailed discussion of the extension and its purpose. The following character set includes the only characters that may be used in a BASIC filename:

the letters A through Z

the numbers zero through nine

the special characters { } ( ) @ # $ % – & ! _ ' ' / ~

Pathnames

Because BASIC 2.0 provides for hierarchial directories, the file specification function in BASIC 2.0 has been expanded to include specification of a *path* to a file. A *pathname* is a sequence of directory names separated by back slashes (\) and followed by a filename. The syntax for a file specification including a pathname is:

device:pathname

BASIC searches the currently loaded disk if no device is specified. If a path begins with a back slash, it starts at the root directory. For example, if your current directory is INVEST, and you want to specify the file IRA, you could use either

```
\INVEST\TAX\IRA
```

or

```
TAX\IRA
```

In the first sample command, the full path from the root was specified by the leading back slash. In the second, the path from the current directory was designated.

Pathnames may be used with the following commands:

BLOAD	CHDIR	LOAD	NAME	RUN
BSAVE	FILES	MERGE	OPEN	SAVE
CHAIN	KILL	MKDIR	RMDIR	

NOTE: If a device name is incorrectly placed, e.g., after the path, the system will display a Bad file name error message.

A path may include no more than 63 characters.

String constants used for paths must be enclosed in quotation marks.

PROGRAM FILE MANAGEMENT

The BASIC commands described briefly below are used to manipulate program files in different ways. A more detailed discussion of each command appears in Chapter 7.

Command	BASIC Action
SAVE *filespec*[,A] or SAVE *filespec*[,P]	Writes to disk the program file currently in memory. The "A" option writes the file in ASCII characters; if option A is not specified, the file is written in packed binary format. The "P" (for Protect) option allows to save a file in an encoded binary format. Since a file saved with option P cannot be listed, saved, edited, or "un-protected", you might wish to save an unprotected copy for listing or editing purposes.
LOAD *filespec*[,R]	Loads the specified program file from disk into memory. The "R" option causes the program to be immediately run upon loading. Before the LOAD command is executed, the current contents of memory are deleted and all files are closed. If option R is included, however, open data files are left open, so as to be available to the running program.

RUN *filespec*[,R]	Loads the specified program file into memory and runs it. Before the LOAD command, the current memory contents are deleted and all files are closed. If the "R" option is included, however, open data files are left open.
MERGE *filespec*	This command loads the specified program file into memory, but does not delete the current memory contents. Instead, the program line numbers on the disk file and those on the program lines in memory are merged. If two lines carry the same number, the line in the disk program is retained. After this process is complete, the MERGED program is in memory, and BASIC returns to the command level.
KILL *filespec*	Deletes the specified file from the disk.
NAME *filespec* AS *filename*	Changes the name of the specified file to *filename*.

DATA FILES

BASIC allows you to create and access two types of disk data file: *sequential* and *random access*. There are three main differences between the two file types: storage requirements, ease of input, and speed of access.

Random access files require less storage space because they are stored on disk in a binary format, while sequential files are stored in ASCII characters.

Sequential files are straightforward to create but cumbersome to access, because the data is stored sequentially as it comes in. To find a piece of information, the computer must start at the beginning and read through the file until it finds the desired data.

To access a random file is a much faster process, because the data is formatted into units called records, and each record has a number that acts as its address. However, creating a random access file is a somewhat more complex process, since a buffer area is used to format the records before they are written to the file, and you must program the sequential numbering of the records.

File Functions

Certain BASIC functions are specifically applied to files, rather than, for instance, variables. These include the LOC and LOF functions.

The LOF function will return the number of bytes of disk space occupied by the file, allowing you to keep track of the file's size relative to the total storage space.

When applied to a sequential file, the LOC function will return the number of data records that have been written to or read from a file since it was last opened. Thus, LOC may be used as a comparator for an end-of-file test.

When applied to a random file, the LOC function returns the number of the "current record", which is the last record number referenced by GET or PUT.

The PRINT # USING statement may be used to write formatted data to a file.

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

This statement would write four numbers designated A, B, C, and D to the file, separated by commas, and with a period so placed that the numbers are expressed with two decimal places. Other examples appear in the example programs.

Creating and Accessing Sequential Files

The following statements and functions are applied to sequential files.

Each is explained in detail in Chapter 7.

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

Two program steps are necessary to create a sequential file:

1. Open the file for output with the OPEN statement:

   ```
   100 OPEN "FILE2" FOR OUPUT AS #1
   ```

   This statement may alternatively be written:

   ```
   100 OPEN "0",#1,"FILE2"
   ```

Later, when you wish to add more data to FILE2, open the file for append, rather than for output, as follows:

```
100 OPEN "FILE2" FOR APPEND AS #1
```

You may append the file also in the following manner:

```
100 OPEN "A",#1,"FILE2"
```

If you open an existing file for output, you will destroy its contents.

2. With the WRITE #, PRINT #, or PRINT # USING statements, write data to the file:

```
200 WRITE #1,DA$,DB$
```

This statement says, "Write to File #1, which I told you in the OPEN statement is FILE2, the current values of these variables."

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

1. Close the file for output purposes using the CLOSE statement, then reopen it for input (to the program) with the OPEN statement:

```
300 CLOSE #1
400 OPEN "FILE2" FOR INPUT AS #1
```

You may reopen the file also in the following manner:

```
400 OPEN "I",#1,"FILE2"
```

2. Read data into the program with INPUT # and LINE INPUT #statements:

```
500 INPUT #1,DX$,DY$
```

This statement says, "Get me the next two units of data in File #1, and assign these names to the values.".

Example Programs for Sequential Files

In the following example, lines 20, 30, and 40 use INPUT statements to specify the prompt the program will make for the data, which will be input from the keyboard. "D$", "N$", and "M$" are the names to be assigned to each field.

```
10 OPEN "FILE2" FOR OUTPUT AS#1
20 INPUT "DATE";D$:PRINT
25 IF D$="00" THEN CLOSE:END
30 INPUT "NAME";N$
40 INPUT "MESSAGE";M$
```

Statement 25 says, "When I enter the word '00' as a date, this means I am finished entering data, so stop the program."

In the next example, The WRITE statement writes three pieces of data to the file. Statement 60 says, "Go back to Statement 20 and continue."

```
50 WRITE #1,D$,N$,M$
60 PRINT: GOTO 20
```

The effect of the RUN command is shown in the following example. The question mark and the words to its left are the computer's display. To the right of the question mark is the user's entry from the keyboard.

```
RUN

DATE? 01/10/84
NAME? J.JONES
MESSAGE? DINNER

DATE? 02/20/84
NAME? S.OLIVER
MESSAGE? MEETING AT 8:30

DATE? 00

OK
```

The following program can be used to access the file just created:

```
10 OPEN "FILE2" FOR INPUT AS #1
15 IF EOF(1) THEN CLOSE: END
```

Line 15, using the EOF function, checks to see if the end of the file has been reached, and if it has, stops reading data. (BASIC marks the end of a sequential file with a "CHR$(26)", so you should not use this character in your file entries.)

```
20 INPUT #1,D$,N$,M$
```

The INPUT # statement is used to read a set of data fields into the program from the file.

```
30 IF LEFT$(D$,2)="02" THEN PRINT N$
```

The IF statement says, "If the leftmost portion of D$, which is two characters long, is '02', then print the name."

```
40 GOTO 15
```

Statement 40 says, "Whatever the outcome of the comparison, continue and examine the next set of data fields."

```
RUN
```

The RUN causes the computer to display:

```
S.OLIVER
```

Creating and Accessing Random Access Files

The following basic statements and functions are applied to random access files. Each is described in detail in Chapter 7.

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

Creating a random file involves the following steps:

1. Open the file for random access with the OPEN statement:

   ```
   100 OPEN "TELEPHONE" AS #1 LEN=32
   ```

   The length of each record is specified as 32 bytes.

2. Allocate space in the random buffer area for each record's data fields, using the FIELD statement:

   ```
   200 FIELD #1, 2 AS NU$, 20 AS N$, 8 AS T$
   ```

   This statement says, "The first two bytes of the record will be referenced as NU$, the next twenty as N$, and the last eight as T$."

3. Move the data into the buffer with the LSET or RSET statements:

   ```
   300 LSET NU$=MKI$(NUMBER)
   400 LSET N$=NAME$
   500 LSET T$=TEL$
   ```

Statement 300 says, "Set NU$ to equal the value represented by NUMBER, expressing the integer value as a numeric string." (Numbers must be made into string fields when they are placed in the buffer. This is done with the "make functions: MKI$ makes an integer into a string, MKS$ changes a single-precision number, and MKD$ is used to convert a double-precision value.) Statement 400 says, "Set N$ to equal the data referenced by NAME$." Statement 500 says, "Set T$ to equal the data represented by TEL$."

4. Write the buffer data record to the file with the PUT statement:

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

The statement say, "Write the record in the buffer to the file, and number it with the integer number referenced by the variable CODE%."

In order to access the data in a random file, it is necessary to perform the following program steps:

1. Open the file for random access:

```
100 OPEN "TELEPHONE" AS #1 LEN=32
```

2. Allocate space in the buffer for the data records that will be read from the file, using the FIELD statement:

```
200 FIELD #1, 2 AS NU$, 20 AS N$, 8 AS T$
```

This statement says, "The records to be read are divided into three fields; the first is two bytes long and is called NU$, the second is twenty bytes long and is named N$, and the third is eight bytes long and named A$."

Note that one OPEN statement and a single FIELD statement per program will usually allow both input and output to the same random file.

3. Move a specified record into the buffer, using the GET statement:

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

The GET statement says, "Retrieve from the file the record whose number is the value represented by CODE%, and write it into the buffer."

4. Since the buffer data may now be used by the program, numeric strings must be converted back to numbers. The "convert" functions accomplish this:

```
400 PRINT N$
500 PRINT CVI (NU$)
```

These statements say, "Print N$ and NU$, expressing NU$ as an integer value". (CVS is used to convert a string to a single precision number and CVD converts a numeric string to a double-precision constant.)

Example Program for Random Access Files

This program writes keyboard input to a random file:

```
10 OPEN "TELEPHONE" AS #1 LEN=32
20 FIELD #1,2 AS NU$, 20 AS N$, 8 AS T$
30 INPUT "2-DIGIT CODE";CODE%
35 IF CODE%=99 THEN CLOSE: END
```

Line 30 causes the computer to display "2-DIGIT CODE?" The user input of a two-digit number will be referenced by the variable name CODE%. Line 35 tests the value of CODE% "99" signifies that the last record has been entered; when the entered value is 99, the files are closed and the program stops. If another code is entered, the program continues:

```
40 INPUT "NUMBER";NUMBER%
50 INPUT "NAME";NAM$
60 INPUT "PHONE";TEL$: PRINT
```

These following statements cause the computer to display prompts for data:

```
NUMBER?
NAME?
PHONE?
```

DISK FILE

The user's reponses will be named, respectively, NUMBER%, NAM$, TEL$. ": PRINT" is just for a carriage return and line feed.

```
70 LSET NU$=MKI$(NUMBER%)
80 LSET N$=NAM$
90 LSET T$=TEL$
```

The buffer variables are set to equal, respectively, NUMBER%, (the integer number being changed to a numeric string) NAM$, and TEL$.

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

The buffer record is written to the file and numbered for referenced with the CODE% value.

```
110 GOTO 30
```

The GOTO statement says, "Go back to Line 30 and do what it says there."

This next program prompts the user to enter a number between 01 and 99, then locates the file record numbered with that code, and displays the information it contains.

```
10 OPEN "TELEPHONE" AS #1 LEN=32
20 FIELD #1, 2 AS NU$, 20 AS N$, 8 AS T$
```

The data file is opened and the buffer fields defined. (As noted above, if this has been done before within the program, these steps are not necessary to read the file.)

```
30 INPUT "2-DIGIT CODE";NUM%
35 IF NUM%=99 THEN CLOSE: END
```

The computer prompts the user for a two-digit number, which is given the variable name NUM% Entering '99' means the user is finished, and the program will end.

```
40 GET #1, NUM%
```

The GET statement says, "Get the record numbered 'NUM%' from FILE #1 and put it in the buffer."

```
50 PRINT USING "NUMBER=###";CVI(NU$)
60 PRINT N$
70 PRINT T$: PRINT
```

The PRINT statements say, "Display the record this way: first the name; then the amount with the first three characters in the string followed by a period, then the last two string characters; then the phone number.

```
80 GOTO 30
```

The GOTO statement says, "Go back to Line 30, and do what it says."

MEMO

CHAPTER 6

BASIC PROGRAMMING ELEMENTS

RESERVED WORDS 6-3

NUMERIC REPRESENTATION IN BASIC 6-4
- Numeric Precision 6-5
- Conversion to a Different Precision 6-5
- Formatting Numeric Output 6-6
- Formatting Double-Precision Numeric Output 6-7

CONSTANTS 6-9

VARIABLES 6-10
- Naming Variables 6-11
- Declaring Variable Types 6-11
- Array Variables 6-12

EXPRESSIONS, OPERATORS, AND FUNCTIONS 6-14
- Arithmetic Operators 6-15
- Relational Operators 6-16
  - Numeric Comparisons 6-17
  - String Comparisons 6-17
- Logical Operators 6-19
- Execution of Numeric Operations 6-22
- String Operators 6-23

PROGRAMMING ELEMENTS

I/O OPTIONS 6-24
Device Name 6-24
Screen Uses 6-25
Text and Graphics Modes 6-26
Specifying Graphics Coordinates 6-27
Window and View 6-27

Additional I/O Features 6-28

PROGRAMMING ELEMENTS

RESERVED WORDS

Certain sequential groups of characters have specific meaning to BASIC, and are called *reserved words*. These may not be used as file or variable names, though they may be embedded within other characters of a name. For instance, "GET" is a reserved word because it is the name of a BASIC command. But it would be allowable to name a program "GOGETREC".

BASIC reserved words are as follows:

> All BASIC keywords that are used as names for BASIC commands, statements functions, and variables are reserved. Each of these appears prominently in UPPERCASE in both the text and table of contents for Chapter 7, where each is described.
>
> In addition, the following words are also reserved:

AND	IMP	OR
DEFDBL	IOCTL	STEP
DEFINT	IOCTL$	THEN
DEFSNG	MOD	TO
DEFSTR	MOTOR	USING
ELSE	NOT	XOR
EQV	OFF	
FNxxxxxxxx		

Reserved words must always be separated from other parts of a BASIC statement by blanks or other special characters permitted by the syntax. This is called "delimiting" them so that the BASIC program line is separated from the line number by a blank. You can also include more than one command in a single program line, but you must delimit each one with a colon, that is, the first character of the new command must be separated from the last in the previous one by a colon.

PROGRAMMING ELEMENTS

NUMERIC REPRESENTATION IN BASIC

Numbers may be positive or negative, may be in either decimal, hexadecimal, or octal mode, and do not include commas. *Decimal numbers* may be expressed as integer, fixed point, or floating point numbers, defined as follows:

* An integer is a whole number from −32768 through 32767 which does not include a decimal point.

* A fixed point number is any real number, including those with decimal points.

* A floating point number is a number expressed with an exponent that functions as in scientific notation. Such a number is represented in three parts:

  the mantissa, which is an integer or fixed point number

  the letter E, except in double-precision numbers where the letter D is used to specify the exponent

  a positive or negative integer that is the exponent

  The following example of a floating point number illustrates both the floating decimal point concept and the way this kind of number is entered and stored:

  ```
  81.79E-7
  ```

  This number, which is read "eighty-one point seven nine times ten to the minus seven", is equivalent to the fixed point number .000008179. "81.79" is the mantissa and "−7" is the exponent.

* A floating point number may be any number from ±2.9E−39 to ±1.7E+38.

PROGRAMMING ELEMENTS

A hexadecimal number may contain up to four hex digits, and is prefixed by "&H". Hex digits are the numbers zero through nine and the letters A, B, C, D, E, and F (10 through 15). As an example, the hex number 46F is represented as:

```
&H46F
```

An octal number may have up to six octal digits, which are the numbers zero through seven, and includes either the prefix "&O" or "&". For example, the octal number 326 could be represented as either:

```
&O326 or &326
```

Numeric Precision

A number that is represented exponentially (that is, a floating point decimal number) may also be defined as either a single- or double-precision number.

A single-precision number may be any fixed point number of seven digits or less, which may either be expressed in exponential form using "E", or appended with an exclamation point. Seven digits will be stored and printed for such a number, but only six will be significant.

A double-precision number may be any fixed point number with eight or more digits, which may either be expressed exponentially using "D" to specify the exponent, or including a trailing number sign (#). BASIC will store 17 significant digits and print as many as 16.

Conversion to a Different Precision

BASIC converts numbers from one precision to another, under certain circumstances and according to the following rules:

If a number of one precision is assigned to a field defined to be of a different precision, the number will be stored as the precision declared for the target field. For example, if a number with a decimal portion is placed in a field defined to contain an integer, then the number will be stored as an integer.

When a number of a higher precision is converted to one of lower precision, it is rounded rather than truncated.

If a lower precision number is converted to a higher precision, the higher precision result will be no more accurate than the lower precision number, since no computation is performed in the conversion. Logical operators (described later in this chapter) convert their operands to integers, and the result is an integer number.

When an arithmetic or relational expression is evaluated, all the operands are converted to the precision of the most precise. Arithmetic operations return a result to this degree of precision.

Formatting Numeric Output

Certain BASIC statements and functions may be used to specify the desired format and accuracy of numeric output.

You may, for instance, use the DEFINT statement (see Chapter 7) to display integer results when the calculations may have been performed with higher-precision numbers:

```
100 DEFINT A
200 A=NUM1#*NUM2#
300 PRINT A
```

The integer portion of the result would be displayed.

The PRINT USING statement (see Chapters 5 and 7) is used to specify formatting of numeric output. For example, a statement such as the following would place commas in numbers:

```
400 PRINT USING "###,###";NUM%
```

Formatting Double-Precision Numeric Output

Use DEFDBL to define your constants and variables as double-precision numbers when formatting output:

```
10 WIDTH 80
20 DEFDBL A
30 A=60#
40 PRINT A/100#,
50 A=A+1
60 IF A<90# GOTO 40

RUN
    •6        •61       •62       •63       •64
    •65       •66       •67       •68       •69
    •7        •71       •72       •73       •74
    •75       •76       •77       •78       •79
    •8        •81       •82       •83       •84
    •85       •86       •87       •88       •89
```

PROGRAMMING ELEMENTS

Use PRINT USING and LPRINT USING when you want the output to be in decimal notation. You can choose the format for the printed or displayed results. This example prints up to four decimals to the left of the decimal point and two decimals to the right:

```
10 WIDTH 40
20 N=950.25
30 PRINT USING "####.##      ";N;
40 N=N+10.5
50 IF N<1100 GOTO 30

RUN
 950.25     960.75    971.25    981.75
 992.25    1002.75   1013.25   1023.75
1034.25    1044.75   1055.25   1065.75
1076.25    1086.75   1097.25
Ok
```

Do not use both single- and double-precision numbers in the same formula as accuracy will be reduced.

You get greater accuracy when you use double-precision transcendentals.

PROGRAMMING ELEMENTS

CONSTANTS

A constant is a value that is used during execution of a program. BASIC uses two types of constants: string and numeric.

A string constant is a sequential string of up to 255 characters that is enclosed in double quotes. A string constant may contain any legal BASIC characters in the form of words, phrases, or numbers. For example:

```
"COMPUTER"
"$18.75"
"Date of Birth"
"Your library book is six months overdue."
```

Note that a numeric string such as "$18.75" cannot be used for computations.

For mathematical operation, BASIC uses numeric constants, whose values are always numbers, as defined in the preceding section of this chapter.

VARIABLES

A variable, like a constant, is a value that is used during program execution, but unlike a constant:

A variable value may change during the course of the program.

A variable is referenced with a unique name.

Variables, like constants, are of two types: string and numeric.

The length of a string variable is determined by the value assigned to it, and may be anywhere from zero to 255 characters. A string variable may only contain a character string.

The value of a numeric variable is always a number.

A variable's value may be set to that of a constant, or its value may be the result of calculations performed, or data entered, during execution. However the value is set, the variable type must be consistent with the type of data assigned to it.

The value of a numeric variable that has not yet had a value assigned to it is zero. A string variable not yet assigned a value is considered to be zero characters long and have a null value.

Naming Variables

A variable name may be as many as 40 characters long. The name may be composed of letters and numbers, and the period may be used. The first character must be alphabetic. The $, %, !, and # symbols are also allowed as the last character of the name, to indicate the variable type (see below).

Variable names may not be reserved words, either alone or appended with $, %, !, or #, but reserved words may be embedded within the name.

Variable name with FN as the first two characters is assumed to call a user-defined function (see the DEF FN statement in Chapter 7).

Declaring Variable Types

A variable's type is specified by the last character of its name, which is called the type declaration character. The characters that are used for this function, and the variable types they designate, are listed below:

- $ string variable
- % numeric integer variable
- ! numeric single-precision variable
- # numeric double-precision variable

A variable name without a type declaration character is considered to represent a single-precision numeric variable.

Though higher precision numbers result in more accurate computations, these variables require more space to store and more time to use. Your requirements and those of your program must be weighted when declaring numeric variable types.

Certain variable types may also be declared with the DEFINT, DEFSNG, DEFDBL, and DEFSTR statements. Refer to Chapter 7 for a detailed discussion of each of these statements.

Array Variables

An array is a group of variables that are referenced by a collective name. Individual array values are called elements. Array elements may be used in any BASIC expression, statement, or function where a variable can be used.

The name of each array element consists of the array name followed by a subscript in parentheses. The subscript indicates the element's position in the array.

Naming an array (thereby declaring the variables' type) and specifying the number and positioning of the elements is called defining the array. This is done with the DIM (Dimension) statement, which is used like this:

```
100 DIM A$(5)
```

This statement sets up an array named A$, which contains six character string elements that will be referred to as A$(0), A$(1), A$(2), A$(3), A$(4), and A$(5). Each element initially has a null value.

Array elements are arranged in rows and columns, both of which may be specified in the DIM statement. If, as in the above example, only the number of rows is defined, the array is said to be a one-dimensional array.

A DIM statement to define a two-dimensional array looks like this:

```
200 DIM A(2,2)
```

This statement creates an array named A that contains nine single-precision variables, each initially equals to zero. The elements are named and arranged like this:

	Column 0	Column 1	Column 2
Row 0	A(0,0)	A(0,1)	A(0,2)
Row 1	A(1,0)	A(1,1)	A(1,2)
Row 2	A(2,0)	A(2,1)	A(2,2)

Use of a subscripted variable for which a corresponding array has not been defined causes BASIC to set up a one-dimensional array with 11 elements. The following statement, lacking an associated DIM statement, would result in the elements in the default array being named NUM#(0) through NUM#(10), and the double-precision value named RESULT# being assigned to array element NUM#(3).

```
500 NUM#(3)=RESULT#
```

This process is called implicit declaration of an array.

EXPRESSIONS, OPERATORS, AND FUNCTIONS

Expressions are used to specify mathematical or logical operations on one or more constants or variables, called "operands" in this context. Operators within expressions designate the operation to be performed. Functions are included in expressions to reference operations defined elsewhere that are to be applied to one or more of the expression's operands.

The following example BASIC statements demonstrate these concepts:

```
500 RESULT=(NUM1+NUM2)-TAN(ANGLE1)
600 IF RESULT>360 GOTO 100
```

In Statement 500, the expression is the portion of the statement to the right of the equal sign; in Statement 600, it is "RESULT > 360". In Statement 500, the expression contains the operators "+" and "−", specifying addition and subtraction; the operator in Statement 600 is ">", which means "greater than". "TAN" in Statement 500 refers to the tangent function, which BASIC is equipped to perform on a give angle.

Many operations may be performed on both numbers and character strings, but if an operator returns a numeric result, it is called a numeric operator. A string operator returns a string result.

In the same way, a numeric function is one applied to a number that returns a numeric result. String functions are applied to character strings, and the result is a string.

Arithmetic Operators

When evaluating a mathematical expression, BASIC performs the following arithmetic operations as needed, in this order:

Operator	Operation	Example	Expression Meaning
^	Exponentiation	A^B	A to the power of B
–	Negation	–A	Minus A, (make A negative)
*	Multiplication	A*B	Multiply A by B
/	Floating Point Division	A/B	Divide A by B
\	Integer Division	A\B	Round A and B to integers in the range of –32768 to 32767; then divide A by B and truncate the quotient to an integer.
MOD	Modulo Arithmetic	A MOD B	Return the integer remainder of an integer division operation.
+	Addition	A+B	Add A and B.
–	Subtraction	A–B	Subtract B from A.

PROGRAMMING ELEMENTS

Relational Operators

A relational operator compares one value to another. These may be either two numeric values or they may be derived via a process defined by an expression.

The following operators are used to specify a comparison for the indicated relationships between values:

Operator	Relationship	Example	Expression Meaning
=	Equal	A=B	A equals B.
<> or ><	Unequal	A<>B	A is not equal to B.
<	Less than	A<B	A is less than B.
>	Greater than	A>B	A is greater than B.
<=or=<	Less than or equal to	A<=B A=<B	A is either less than or equal to B
>=or=>	Greater than or equal to	A>=B A=>B	A is either greater than or equal to B.

If the comparison as written represents a true statement, then the result of the comparison is stored as "true" (binary −1). If the comparison, as a statement, is not true, then a "false" result is stored (zero).

Relational comparisons are usually performed as part of an IF statement (see Chapter 7), where the result of the comparison determines a decision about the direction of the program's flow. For example:

```
100 IF X>CODE GOTO 500
200 GOTO 10
```

This statement says, "If X is greater than CODE, then continue executing at Statement 500. If this is not the case, continue with the following statement, 200."

Or the result may be output, either explicitly or in the form of a code. For instance:

```
200 IF AMOUNT>LIMIT THEN PRINT "OVER LIMIT" ELSE PRINT
    "OKAY"
```

BASIC will compare AMOUNT to LIMIT, and if AMOUNT is larger, will display the words "OVER LIMIT" (without the quotes). If AMOUNT is not more than LIMIT, the word "OKAY" will be displayed. The following statement, however, uses the result differently.

```
200 PRINT AMT>LMT
```

This statement say, "Compare AMT to LMT and print the result." If AMT is greater than LMT, BASIC will display "−1" to indicate "true". If not, a zero will be displayed, indicating a "false" result.

Numeric Comparisons

When an expression contains both relational and arithmetic operators, the arithmetic is done first, so that the comparison can be made between two values. For instance, in the following expression the values (A plus B) and (NUM divided by X) are computed, then these computed values are compared:

```
A+B=NUM/X
```

If the values are equal, the result is "true". If not, a "false" result is stored.

String Comparisons

Though relational operators produce numeric results (−1 or zero) and are thus considered to be numeric operators, they are also used to make comparisons between character strings.

When a string comparison is performed, BASIC successively examines the ASCII codes for each corresponding character position in both strings. The strings are considered equal if all the codes in each are the same. Differing codes, at any point, signal unequal characters, and the relationship between them is deduced by a process similar to alphabetizing.

For example, in strings of equal length, B is considered "greater than" A, because A precedes B in the alphabet. Thus, the string "XY" would be "less than" the string "YZ". And, since a lowercase letter is considered to be "greater than" its uppercase counterpart, the following comparison would be true:

```
"cp">"CP"
```

(Note that when a string constant is used in a comparison expression, it must be enclosed in quotes.) Numbers are considered "less than" letters, so the following comparison would be false:

```
"FILE1">"FILEA"
```

A string constant can also be compared to the contents of a string variable, such as:

```
NAME$="H. PHILIP LIMBACHER"
```

When two strings of unequal length are compared, the shorter string is considered "less than" the longer.

Logical Operators

A logical operation combines the results of two or more relational comparisons to make a decision. Logical operators use the "true" or "false" results of the comparisons as operands to calculate a "true" or "false" result of the logical operation.

The logical operators are as follows, listed in the order they are performed if more than one appears in a single statement:

Operator	Operation
NOT	Logical Complement
AND	Conjunction
OR	Disjuction
XOR	Exclusive Or
EQV	Equivalence
IMP	Implication

The NOT operator is used in two ways. In decision-making, this operator can specify a branch in the program flow if an expression is false, for example:

```
400 If NOT (CODE=100) THEN 200
```

This statement says, "If CODE does not equal 100, continue at Statement 200. If it does, continue with the next statement." NOT may also be used to change the sign of a value to that of its complement.

```
500 CODE=NOT CODE
```

This statement says, "Set the value of CODE to 1's complement." For example, if CODE were equal to 50, it would be set to −51; if it were −80, it would be set to 79.

The result of a NOT operation is arrived at in the following way: if the relational expression is true, the logical result will be "false"; if the expression is false, the result of the NOT operation will be "true". This method of deducing a result is called "NOT logic".

AND returns a "true" result only if both of two expressions are true. For example:

```
200 IF CODE>50 AND NUM=100 THEN 500
```

If CODE is greater than 50, and if NUM equals 100, then the program will continue at Statement 500. But if either situation is not the case, the result of the logical AND operation will yield a "false" result, and the program will continue at the next instruction.

The following table shows how "AND logic" derives a result from the four possible combinations of "true" and "false" that can result from the comparison of two expressions.

Expression 1 Result	Expression 2 Result	AND Result
T	T	T
T	F	F
F	T	F
F	F	F

OR will return a "true" result if either of two expressions is true, for example:

```
200 IF CODE=100 OR NUM>20 THEN 500
```

If CODE is 100, or if NUM is greater than 20, the program will continue at Statement 500. Only if neither expression is true the OR operations will give a "false" result, causing the program to continue on the following line.

The table below shows how the results of a relational comparison are combined with "OR logic" to produce a result.

Expression 1 Result	Expression 2 Result	OR Result
T	T	T
T	F	T
F	T	T
F	F	F

XOR returns a "true" result when one expression is true and the other false. If both are either true or false, the logical result will be "false". "XOR logic" derives a result as follows:

Expression 1 Result	Expression 2 Result	XOR Result
T	T	F
T	F	T
F	T	T
F	F	F

EQV returns a "true" result if both expressions are either true or false. If the results of the realtional comparison differ, the logical result will be "false". "EQV logic" arrives at a result in the following way:

Expression 1 Result	Expression 2 Result	EQV Result
T	T	T
T	F	F
F	T	F
F	F	T

PROGRAMMING ELEMENTS

The result of an IMP operation will be "true" unless the first expression is true and the second is false. "IMP logic" reaches a result as shown below.

Expression 1 Result	Expression 2 Result	IMP Result
T	T	T
T	F	F
F	T	T
F	F	T

Execution of Numeric Operations

To summarize, BASIC performs numeric operations in the following sequence:

1. First, any specified *functions* are performed.

2. Then, *arithmetic operations* are done in the order given in the section titled "Arithmetic Operators", which is the standard mathematical order of operations.

3. *Relational operations* are evaluated next.

4. Last, *logical operations* are performed in the order given in the preceding section, "Logical Operators".

If an operation is specified more than once in a statement, each occurence is executed in left-to-right sequence.

Operations enclosed in parentheses are done first in the order listed above.

String Operators

An operation known as *concatenation* is used to combine existing character strings into a new string. Concatenation is considered a string operation because, unlike the operations described previously in this chapter, the result of the operation is a character string, rather than a numeric value.

The plus sign is the operator used to specify concatenation (joining together) of two or more character strings. The following example would print three fields as a unit:

```
10 CITY$="San Francisco,"
15 STATE$=" CA"
20 ZIP$=" 94117"
50 PRINT CITY$+STATE$+ZIP$
RUN
San Francisco, CA 94117
```

Other string operations are performed by functions built into BASIC. Refer to Chapter 7 for a detailed description of the purpose and use of all BASIC string functions.

I/O OPTIONS

Device Name

The device name consists of up to four characters and a colon (:). Device names and meanings are listed below.

Device	Meaning
KYBD:	Keyboard. (Input)
SCRN:	Screen. (Output)
LPT1:	First printer. (Internal Printer) (Output or random)
LPT2:	Second printer. (Output or random)
LPT3:	Third printer. (Output or random)
COM1:	Internal Serial (RS232C) Interface (Input and output)
COM2:	Optional Serial Interface (Input and output)
A:	First floppy disk drive. (Input and output)
B:	Second floppy disk drive. (Input and output)
C:	First hard disk drive. (Input and output)
D:	Second hard disk drive. (Input and output)

Refer to "Naming Files" in Chapter 5.

If your system supports more than two floppy disks, the device names change slightly. For example, if you have three floppy disk drives, the third disk becomes C. Now the first hard disk is named D, and the second one is E. If you are using four disk drives, the last disk drive will be named D. The first hard disk becomes E, and the second one is F.

Screen Uses

The Sr. Partner can display black and white output on the built-in display, or color if you use a RGB monitor. These colors can be displayed as text composed of standard ASCII characters, or in the form of points, lines, and figures.

If you use a RGB monitor, text may be displayed either in black and white, or in up to 14 additional colors. All 16 available colors are listed with the associated numbers used to specify them as part of the description of the COLOR statement in Chapter 7. The COLOR statement may also be used to specify the color of the "border screen". This is the area around the portion of the screen that is used for the display. Additionally, Sr. Partner makes available a extensive and versatile graphics capability, and also allows the division of the screen buffer area into eight sections (called "pages") that may be individually written to or displayed. This is done with the SCREEN statement (refer to Chapter 7).

Text and Graphics Modes

When BASIC displays output using the standard ASCII characters, it is said to be in *text mode*. Text is displayed using 25 horizontal screen lines. The line at the top of the screen is Line 1, and the bottom line is Line 25. Each line will be either 40 or 80 characters long, depending on how you have specified the screen width. Each character position is numbered, beginning with Position 1 on the left side of the screen.

Line 25 is normally reserved for output generated by the "soft keys" (see Chapter 4). When more output follows Line 24, the screen is "scrolled". This is what happens: Line 1 disappears, Lines 2 throughout 24 move up one line each to become Lines 1 through 23, and the next line of output is displayed as Line 24.

In *graphics mode*, text may be displayed, as well as figures drawn with two graphic resolutions:

With *high resolution*, you can reference 640 horizontal and 200 vertical points on the screen. The points are addressed by coordinate numbers, which are assigned sequentially from the upper left corner of the screen, which is (0,0), across each line to the lower right corner point, which is numbered (639, 199).

Text is displayed in high resolution on an 80-character line. Both text and graphics are displayed in two colors. The "foreground" color (that of the character) is white, and the "background" color (that surrounding the character) is black.

With *medium resolution*, you may reference 320 horizontal and 200 vertical screen points, and display output in four colors.

The medium resolution coordinates are numbered (0,0) to (319,199) in the same way as in high resolution.

Text displayed with medium resolution appears on twenty-five 40-character lines. The foreground will be Color Three and the background Color Zero. (Refer to COLOR statement.)

Specifying Graphic Coordinates

A graphic statement must include information that specifies where you want to draw on the screen. This information is supplied in the form of *coordinates* which specify the location of a point on the screen. Coordinates are written in the form of (x,y), where x is the horizontal position any y represents the vertical location. This form of indicating a coordinate point is called "absolute form" because it specifies a position without reference to any others.

Coordinates may also be written using "relative form", which specifies a position relative to the point most recently referenced. This form is written like this:

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

In the above example, x and y are the coordinates of the last referenced point, and the "offsets" are the distances from the previous x and y to the new x and y. How each graphic statement sets the "last point referenced" is explained as part of the description of each statement in Chapter 7. Below is an example of how two points may be defined using absolute and relative form.

```
200 PSET (65,180)
250 PSET STEP (50,-10)
```

Statement 200 uses absolute form to specify a point with coordinates (65,180). Statement 250 uses relative form to specify the next location as a point whose horizontal position is 115 (65+50) and whose vertical position is 170 (180−10); this coordinate would be written (115,170).

Window and View

In BASIC 2.0, there are two coordinate systems, logical coordinates and physical coordinates.
Physical coordinates are the coordinates of the screen, the upper left hand corner is (0,0) and the lower right hand corner is (639,199) or (319,199).
Logical coordinates are programmer defined coordinates using the WINDOW statement.

Additional I/O Features

A number of other features are available to supplement BASIC programming on the Sr. Partner.

Two informational system variables may be initialized and referenced:

> DATE$ gives the date in the form mm-dd-yyyy. For example, November 20, 1984 would be displayed as 11-20-1984.
>
> TIME$ gives the time by the 24-hours system clock in the form hh:mm:ss. For example, 14:15:33 is 2:15 PM, and 09:15:01 is slightly after 9:15 AM.

The Sr. Partner can also be instructed to make sounds with the following statements:

BEEP makes the speaker beep.

SOUND produces a sound of a specified frequency and duration.

PLAY makes the computer play music specified by a character string.

If your Sr. Partner is equipped with a Game Control Adapter, you may use joysticks to play computer games. The system will support either two x- and y-coordinate joysticks, or four one-dimensional paddles, each with a button. The STICK, STRIG, STRIG(n), and ON STRIG statements and functions are applied specifically to joystick applications. Refer to Chapter 7 for a description of these commands.

CHAPTER 7:

COMMANDS, FUNCTIONS, STATEMENTS, AND VARIABLES

This chapter describes every BASIC command, statement, function, and variable. The description of each BASIC component is divided into four sections: Syntax, Purpose, Comments, and Examples.

The section titled *Syntax* shows how the component is structured when it is used. The generalized format should be interpreted as follows:

Words that appear in UPPERCASE are BASIC keywords, and must be used where and as shown, though they may be entered in either capitals or lowercase letters.

Italic portions represent items supplied by you; these are referred to as "parameters" or "arguments" and may be represented by words, such as *line* or *filespec*, or by symbols such as those below, which are used to represent expressions of the indicated types:

x, y, z	(numeric)
i, j, k, l, m, n	(integer)
x$, y$	(string)

v and *v$* are used to symbolize numeric and string variables.

Optional items are enclosed in square brackets ([ ]). Items that may be repeated are followed by an ellipsis (...). Any other punctuation (eg: commas, parentheses, semicolons, hyphens, equal signs, etc.) is part of the syntax and must be included.

The generalized syntax specification usually shows functions and variables on the right side of an assignment statement, ie: *v*=SIN(*x*). This formatting is intended to emphasize the difference in usage between these components and that of commands or statements, not to suggest that they can only be used in assignment statements. In fact, BASIC functions and variables can be used in all ways as regular variables, except that they may not be placed on the *left* side of an assignment statement.

Purpose very briefly describes why the component is used, in terms of what it will accomplish.

The *Comments* explain more fully how to use each component. *Italicized* items in the syntax specification are described in terms of what they represent; the acceptable range, type, or size of each value to be supplied is specified. The Comments also describe any prerequisites to using the components, as well as the effects of executing a statement or command, calling a function, or referencing a variable.

Each description ends with one or more *Examples*, which illustrate how the component may be used to accomplish a given end. Program extracts, as well as sample direct mode statements, are used to clarify both how each component may actually be entered, and how it fits into the context of other associated BASIC processes.

ABS Function

Syntax: *v*=**ABS**(*x*)

Purpose: Returns the absolute value of the expression *x*. The ABS function converts all values of *x* to positive numbers by telling the computer to ignore the sign of *x*.

Comments: *x* must be a numeric expression. The absolute value of a number is positive or zero.

Examples:

```
PRINT ABS(3*(-2))
 6
Ok
```

The absolute value of −6 is printed as a positive 6.

ASC Function

Syntax: *v*=**ASC**(*x$*)

Purpose: Returns the ASCII code of the first character of the string *x$*.

Comments: The ASCII (American Standard Code for Information Interchange) standard character set consists of two-hundreds and fifty-six characters: numbered 0 through 255. (See Appendix C for a list of ASCII values.)

Use of the ASC function results in a numerical value that is the ASCII code of the first character of the string *x$*.

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

The CHR$ function performs the inverse of the ASC function by converting the ASCII to a character.

Examples:

```
100 X$="BIRD"
110 PRINT ASC(X$)
RUN
 66
Ok
```

The ASCII code of capital B is 66. The example below produces the same result:

```
PRINT ASC("BIRD")
```

COMMANDS STATEMENS

ATN Function

Syntax: *v*=**ATN**(*x*)

Purpose: Returns the arctangent of *x* in radians.

Comments: The expression *x* may be any numeric type, however, the evaluation of ATN is always performed in single precision.

Use the ATN function to return the angle whose tangent is *x*. The result falls between –PI/2 to PI/2, where PI=3.141593.

57.29578 (=180/PI) degrees equals one radian. You can convert radians to degrees, by multiplaying by 180/PI.

Examples:

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

Calculates and prints the arctangent of 3.

AUTO Command

Syntax: **AUTO** [*number*] [, [*increment*]]

Purpose: Generates a line number automatically each time you press the <ENTER> key.

Comments: The AUTO command frees you from entering line numbers.

number is the number that is used to begin the line numbering. You may use a (.) in place of the line number to begin numbering with the current line.

increment is the value that is added to each line number to arrive at the next line number.

Numbering automatically starts at *number* and increments each subsequent line number by *increment*. The default for both *number* and *increment* are 10.

If *number* is followed by a comma, and *increment* is not specified, the last *increment* specified in an AUTO command is used.

If *number* is omitted, but *increment* is used, line numbering begins with 0.

If AUTO creates a line number that is already being used, an asterisk is displayed after the number to warn you that anything you enter on this line will replace the existing line. However, you can save the existing line and create the next line number by pressing the <ENTER> key immediately after the asterisk.

AUTO terminates when you press the <Ctrl> <Break> keys. The line in which the <Ctrl> <Break> keys are pressed is not saved. After you enter a <Ctrl> <Break>, BASIC returns to command level.

AUTO is generally used for entering new program lines.

Examples:

```
AUTO
```

This example creates line numbers beginning with 10 and in increments of 10 (10, 20, 30, 40, ...).

```
AUTO 50, 20
```

This command creates line numbers beginning with 50 and in increments of 20 (50, 70, 90, 110, ...).

```
AUTO 100,
```

This creates line numbers beginning with 100 and in increments of 20 because 20 was the increment in the previous AUTO command (100, 120, 140, 160, ...).

```
AUTO ,50
```

This command creates line numbers beginning with 0 because no line number was specified and in increments of 50 as specified (0, 50, 100, 150, ...).

BEEP Statement

Syntax: **BEEP**

Purpose: Produces a single beep from the speaker at 800 Hz (800 cycles per second) for one-quarter of a second.

Comments: You can use BEEP to get the attention of the system operator.

PRINT CHR$(7) has the same effect as BEEP.

Examples:

```
1240 IF X > 12 THEN BEEP
```

This example checks to see if X is out of range. If it is, the computer beeps.

BLOAD Command

Syntax: **BLOAD** *filespec* [,*offset*]

Purpose: Loads a binary file specified by *filespec* into user memory.

Comments: *filespec* is a string expression containing the device and file name.

offset is a numeric expression within the range of 0 to 65535. This is an offset into the segment you declared in the last DEF SEG statement. Before you use BLOAD, execute a DEF SEG statement. When you specify the *offset,* the last DEF SEG address is used.

If *offset* is not specified in BLOAD the *offset* specified at BSAVE is used; that is, the file is loaded into the same location it was saved from.

When the BLOAD is executed, the file specified by *filespec* is loaded into user memory beginning at the specified location.

BLOAD and BSAVE are used to load and save machine language programs. However, the usefulness of BLOAD and BSAVE is not limited to machine language programs. You may specify any segment as the source or target for these commands by using the DEF SEG statement. This permits the video screen buffer to be read from or written to a disk. The BLOAD and BSAVE commands are useful in saving and displaying graphics images.

*** **WARNING** ***

BASIC does not perform an address range check; that is, it is possible to BLOAD anywhere in memory. Be sure you do not BLOAD over BASIC's variable area, or your BASIC programs.

Examples:

```
100 REM Lines 100 through 170 load the screen
110 REM buffer
120 DEF SEG= &HB800
130 REM Line 120 points the segment at the
140 REM screen buffer
150 BLOAD "IMAGE",0
160 REM Line 150 loads the file named IMAGE into
170 REM the screen buffer
```

Using the DEF SEG statements and specifying *offset* at 0 guarantee that the correct address is used.

The example in the BSAVE command description illustrates how the file named IMAGE is saved.

BSAVE Command

Syntax: **BSAVE** *filespec, offset, length*

Purpose: Saves portions of user memory on the specified device.

Comments: *filespec* is a string expression containing the device and file name.

offset is a numeric expression within the range of 0 to 65535. This is an offset into the segment you declared in the last DEF SEG statement. The save will begin from this position.

length is a numeric expression within the range of 1 to 65535. This specifies the length of the memory image to be saved.

BLOAD and BSAVE are used to load and save machine language programs. However, the usefulness of BLOAD and BSAVE is not limited to machine language programs. You may specify any segment as the source or target for these commands by using the DEF SEG statement. This permits the video screen buffer to be read from or written to a disk. The BLOAD and BSAVE commands are useful in saving and displaying graphics images.

Examples:

```
100 REM Lines 100 through 170 save the screen
110 REM buffer
120 DEF SEG= &HB800
130 REM Line 120 points the segmant at the screen
140 REM buffer
150 BSAVE "IMAGE",0,16384
160 REM Line 150 saves the screen in a file named
170 REM IMAGE
```

The DEF SEG statement is used to set up the segment address at the beginning of the screen buffer. Specifying an *offset* of 0 and a *length* of 16384 saves the entire 16K screen buffer.

CALL Statement

Syntax: **CALL** *numvar* [(*variable* [,*variable*]...)]

Purpose: Calls an assembly (or machine) language subroutine.

Comments: *numvar* is the name of a numeric variable, it must contain the starting address of the subroutine as an offset into the current segment of memory. The current segment is defined by the last DEF SEG statement.

variable is the name of a variable passed to the assembly (or machine) language subroutine as an argument.

The CALL statement is one of two ways to interface assembly (or machine) language programs with BASIC. The USR function may also be used. However, the CALL statement can pass multiple arguments.

Examples:

```
100 IN=&HD000
110 CALL  IN(A,B$,C)
120 REM variables A,B$, and C are passed as argu-
130 REM ments to the assembly language subroutine
```

CDBL Function

Syntax: v=**CDBL**(x)

Purpose: Converts x to a double-precision number (CDBL stands for Convert DouBLe).

Comments: x must be a numeric expression.

For converting numbers to single-precision and integer, see the CSNG and CINT functions.

Examples:

```
100 A=454.67: REM Note the number of decimal places
110 PRINT A;CDBL(A)
RUN
 454.67   454.6700134277344
Ok
```

This program prints a double-precision version of the single-precision value stored in the variable named A.

The value of CDBL(A) is only accurate to the second decimal place after rounding because only two decimal places of accuracy were supplied with A. The last 11 numbers, therefore, have no meaning in this example.

CHAIN Statement

Syntax: **CHAIN** [MERGE] *filespec* [,[*line*][,[ALL] [,DELETE *range*]]]

Purpose: Calls a program and passes (chains) variables to it from the current program.

Comments: *filespec* is the name of the program that is called to be chained to. For example:

```
CHAIN "A:ACCT1"
```

line is a line number or an expression that corresponds to a line number in the called program. It specifies the starting point for execution of the called program. For example:

```
CHAIN "A:ACCT1",400
```

In this example, program ACCT1 in drive A begins executing at 400.

If *line* is omitted, execution begins at the first line of the called program.

line is not affected by a RENUM command.

Therefore, if ACCT1 is renumbered, this example CHAIN statement should be changed to point to the new line number. The line numbers in range, however, are affected by a RENUM command.

ALL specifies that every variable in the current program is passed (chained) to the called program. For example:

```
CHAIN "A:ACCT1",400,ALL
```

If the ALL option is omitted, you must include a COMMON statement in the current program. The COMMON statement will list the variables to be passed.

MERGE brings a subroutine into the BASIC program as an overlay. (MERGE overlays the current program with the called program.)

> **NOTE:** The called program must be an ASCII file if it is to be merged. For example:
>
> ```
> CHAIN MERGE "OVRLAY1",400
> ```

After the MERGE (overlay) is executed and used for a specific purpose, it is usually desirable to delete it so that a new overlay may be used. Use the DELETE statement to delete the overlay. For example:

```
CHAIN MERGE "OVRLAY1",400, DELETE 400-2000
```

This example deletes lines 400 through 2000 of the current program before loading in the called (overlay) program.

Notes:

1. The CHAIN statement leaves files open.

2. If a different OPTION BASE is set in the current program and the called program, and the current program contains an array and CHAIN statement with the ALL option, or COMMON statement, then the error message "Duplicate Definition" will be displayed.

3. Without MERGE, CHAIN does not preserve variable types or user-defined functions for use by the called program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statement containing shared variables must be restated in the called program.

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

4. The CHAIN statement performs a RESTORE before running the called program. The next READ statement accesses the first item in the first DATA statement found in the program. The result is that the read operation does not continue where it left off in the current program.

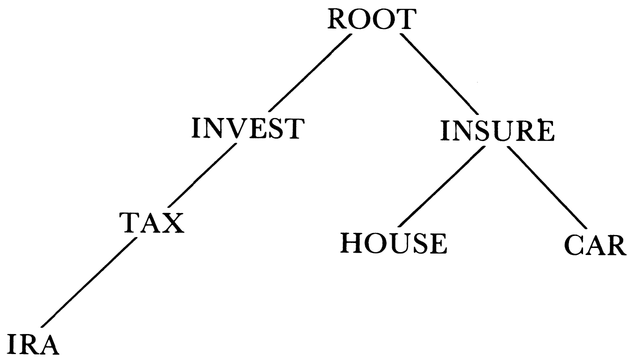
CHDIR Statement

Syntax: **CHDIR** *path*

Purpose: Changes the current directory. For use in BASIC 2.0 and later only.

Comments: *Path* is a valid string expression naming the new directory to be changed to the current directory. The string must not exceed 63 characters.

Examples:



COMMANDS STATEMENTS

These examples are taken from the directory illustrated on the previous page.

Change to the root directory (from any sub-directory).

```
CHDIR "\"
```

Change to the directory IRA from the root directory.

```
CHDIR "INVEST\TAX\IRA"
```

Change to the directory CAR from the directory INSURE.

```
CHDIR "CAR"
```

Change from the directory TAX to the directory INVEST.

```
CHDIR ".."
```

Make INVEST the current directory on the current drive. Make LOANS the current directory on drive B.

```
CHDIR "INVEST"
CHDIR "B:LOANS"
```

LOANS must exist on drive B. While you remain in this structure, *filespec* on drive A refers to the files in the directory INVEST. *filespec* on device B refers to the files in the directory LOANS.

COMMANDS STATEMENS

CHR$ Function

Syntax: *v$*=**CHR$**(*n*)

Purpose: Converts an ASCII code to its equivalent character.

Comments: *n* is a number from 0 through 255.

CHR$ is commonly used to send a special character to the screen or printer. For instance, you can send CHR$(7) to sound a beep through the speaker as a preface to an error message (instead of using the BEEP statement), or you can send a form feed, CHR$(12), to the printer.

The ASC function performs the inverse of the CHR$ function by converting a character to its ASCII code.

See Appendix C for a list of ASCII values.

Examples:

```
PRINT CHR$(84)
T
Ok
```

This example prints the character for the ASCII code 84, which is the upper-case letter T.

CINT Function

Syntax: *v*=**CINT**(*x*)

Purpose: Converts any numeric expression to an integer.

Comments: *x* must be within the range of −32768 to 32767. If it is not, an "Overflow" error occurs.

x is converted to a whole number (integer) by rounding the fractional portion.

See the CSNG and CDBL functions for converting numbers to single-precision and double-precision. See also the FIX and INT functions, both of which return integers.

Examples:

```
PRINT CINT(13.67)
 14
Ok
```

```
PRINT CINT(-13.67)
-14
Ok
```

Observe how rounding occurs in these examples.

CIRCLE Statement

(BASICA only)

Syntax: **CIRCLE** (*x*,*y*),*r*[,*color*[,*start*,*end*[,*aspect*]]]

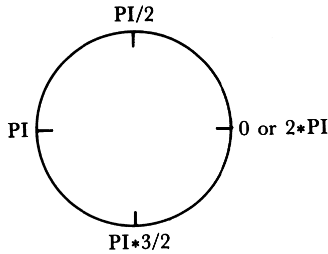
Purpose: Draws an ellipse on the screen. Graphics mode only.

Comments: (*x*,*y*) determine the coordinates of the center of the ellipse. The (*x*,*y*) coordinates may be given in either absolute or relative form. Refer to "Specifying Graphic Coordinates" on page 6-27.

r is the radius of the major axis, in points, of the ellipse.

color is a number (in the range 0–3) that designates the color of the ellipse. In medium resolution mode, the color is selected from the current palette as defined by the COLOR statement. 0 is the background color and 3 is the foreground color. The foreground color is selected if you do not specify this option. In high resolution, the *color* option may be selected as 0 (black) or 1 (white). White is selected if you do not specify this option.

start, *end* are angles in radians and may range from −2*PI to 2*PI (PI=3.141593). *start* and *end* designate where the drawing of the ellipse will begin and finish.



If you use a negative *start* or *end* angle (−0 is not allowed), the ellipse will be connected to the center point with a line, and the angles will be treated as if they were positive. (This is not the same as adding 2*PI.) The start angle may be greater or less than the finish angle.

aspect is a numeric expression that affects the ratio of the x-radius to the y-radius. *aspect* is automatically set to 5/6 in medium resolution and 5/12 in high resolution. These values produce a visual circle given the standard screen aspect ratio of 4/3. The radius is measured in points in the horizontal direction; therefore, if *aspect* is:

less than one, *r* is the x-radius

greater than one, *r* is the y-radius

Examples:

```
100 SCREEN 1: REM specifies a medium resolution
110 REM graphics mode
120 CIRCLE (160,100),80,,,,7/18
130 REM This example draws an ellipse. Run the
140 REM example to see the shape.
```

```
100 PI=3.141593
110 SCREEN 1
120 CIRCLE (160,100),80,,-PI,-PI/4
130 REM This example draws part of a circle.
140 REM Run the example to see the shape.
```

```
100 SCREEN 1
110 CIRCLE (240,150),30
120 REM Draws a circle in the lower right-hand
130 REM corner of the screen.
```

Notes:

1. CIRCLE statement is available in graphics mode. Specify SCREEN 1 (medium resolution) or SCREEN 2 (high resolution).

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

3. CIRCLE will not draw points that are outside the boundaries of the screen.

CLEAR Command

Syntax: **CLEAR** [,[*n*][,*m*]]

Purpose: Sets all numeric variables to zero and all string variables to null, and closes all open files. Options for the CLEAR command set the end of memory and the amount of stack space for use by BASIC.

Comments: *n* is a size (byte) of the BASIC workspace where your program and data are stored, along with the interpreter workarea.

m establishes stack space for BASIC. Stack space is initially 512 bytes.

CLEAR frees all memory used for data without erasing the program in memory at that time. After a CLEAR command:

arrays are undefined

numeric variables are set to zero

string variables are set to null

information set with any DEF statement is lost. (This includes DEF FN, DEFINT, DEFDBL, DEFSNG, and DEFSTR.)

any sound that is running is turned off and reset to Music Foreground

STRIG is reset to OFF

Examples:

```
CLEAR
```

This example clears all data from memory but does not erase the program.

```
CLEAR ,16384
```

In this example, data is cleared and maximum workspace size is set to 16K bytes.

```
CLEAR ,,2500
```

This example clears the data and sets the size of the stack to 2500 bytes.

```
CLEAR ,16384,2500
```

This last example clears the data, sets the maximum workspace for BASIC to 16K bytes, and sets the stack size to 2500 bytes.

Notes:

The following notes provide some instances in which you might want to use the CLEAR command and some of its options:

1. You will need to use the CLEAR command if you wish to use the same array to store two different sets of information at two different points in your program. After the first use, execute the CLEAR command. Now you are prepared for the second use of the array.

2. You will probably want to use the *n* option if you need to reserve space in storage for machine language programs.

3. You may want to use the *m* option if you use a lot of nested GOSUB statements or FOR...NEXT loops in your program, or if you use PAINT to do complex scenes.

CLOSE Statement

Syntax: **CLOSE** [[#]*filenum*[,[#]*filenum*]...]

Purpose: Ends I/O to a device or file.

Comments: *filenum* is the number under which the file was opened in the OPEN statement.

The association between a particular file or device and its file number ends when the CLOSE statement is executed. The file or device may be reopened using the same or a different file number. The file number specified in the CLOSE statement may be reused to open any device or file.

If you execute a CLOSE on a file or device opened for sequential output, the final buffer will be written to the file or device.

If you don't specify file numbers, all opened devices and files will be closed.

If you execute an END, NEW, RESET, SYSTEM, or RUN (without the R option), all open files and devices will be automatically closed. The STOP statement does not close files or devices.

Examples:

```
100 CLOSE
```

This example closes all open files and devices.

```
100 CLOSE #3,5,#7
```

COMMANDS STATEMENS

This example closes the files and devices associated with the file numbers 3, 5, and 7. Notice that the # symbol is optional; use it or not as you prefer.

CLS Statement

Syntax: **CLS**

Purpose: Clears the entire screen or the screen buffer, depending on the current mode.

Comments: If the screen is in text mode, the active page is cleared to the current background color.

If the screen is in graphics mode, medium or high resolution, the whole screen buffer is cleared to the current background color.

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

You may also clear the screen by using the SCREEN or WIDTH statements, or by pressing the Ctrl-Home keys.

Examples:

```
100 CLS
```

This example clears the entire screen or screen buffer, depending on the current mode.

COLOR Statement (In Text Mode)

Syntax: **COLOR** [*foreground*] [,[*background*] [,*border*]]

Purpose: Sets the colors for the foreground, background, and border areas of the screen in the text mode.

Comments: *foreground* is a numeric expression from 0 through 31, designating the character color.

background is a numeric expression from 0 through 7, designating the background color.

border is a numeric expression from 0 through 15, designating the border color.

The following colors are available for *foreground*:

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

COMMANDS STATEMENS

You will notice variation in the colors and their intensity, depending on your display device.

If you set *foreground* to 16 plus the number of the color you want (values 16 through 31), the characters will blink. For example, if you set *foreground* to 29 you will get blinking light magenta.

Only colors 0 through 7 are available for *background*.

Notes:

1. You may create invisible characters, making the foreground color equal to the background color. By changing either the foreground or background color, subsequent characters will be visible.

2. If you omit any parameter, the old value is used for that parameter.

Examples:

```
100 COLOR 13,2,1
```

This example produces a light magenta foreground, a green background, and a blue border screen.

COLOR Statement (In Graphics Mode)

Syntax: **COLOR** [*background*] [,[*palette*]]

Purpose: Sets the background and palette colors in the graphic mode, medium resolution only.

Comments: *background* is a numeric expression that specifies the background color. The colors you can use for *background* are 0 through 31 (see the COLOR statement for text mode).

palette is a numeric expression that specifies your choice of palette colors.

You can select the following palette colors:

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

In palette 0, the colors associated with numbers 1, 2, and 3 are Green, Red, and Brown, respectively.

In palette 1, the colors associated with numbers 1, 2, and 3 are Cyan, Magenta, and White respectively.

You may select a background color that is the same as a palette color.

The *background* and *palette* parameters may be omitted from the COLOR statement. In this case, the old values are used for the omitted parameters.

In graphics mode, the COLOR statement designates the background color and one palette (three colors). The PSET, PRESET, LINE, CIRCLE, PAINT, and DRAW statements may then select any of these four colors for display.

The COLOR statement has meaning in medium resolution only. If you use it in high resolution you will get an "Illegal function call" error.

Values used outside the range of 0 to 255 result in an "Illegal function call" error, but previous values are retained.

Examples:

```
100 SCREEN 1
110 COLOR 8,1
```

Sets the background color to gray, and selects palette 1.

```
120 COLOR ,0
```

The background color stays gray, and palette changes to 0.

COM(n) Statement

(BASICA only)

Syntax:

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

Purpose: Controls (enables and disables) the trapping of communications activity to the internal serial interface or optional serial interface (RS232C, Asyncronous or Communication) adapter.

Comments: *n* is the number (must be 1 or 2) of the serial interface. 1 is the internal serial interface and 2 is an optional serial interface.

The COM(*n*) statement has three forms.

The COM(*n*) ON statement enables communications event trapping by an ON COM statement (see "ON COM Statement," page 7-145). While trapping is enabled, and if a non-zero line number is specified in the ON COM statement, BASIC checks between every statement to see if activity has occurred on the communications channel. If it has, the ON COM statement is executed.

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

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

COMMON Statement

Syntax: **COMMON** *variable*[,*variable*]...

Purpose: Passes variables to a program chained to your current program.

Comments: *variable* is the name of a variable that is to be passed to the called program. Array variables are distinguished from non-array variables by appending "()" to the variable name.

The COMMON statement is used in concern with the CHAIN statement. Although COMMON statements may be anywhere in a program, it is recommended they be at the beginning. An unlimited number of COMMON statements may appear in a program, however, the same variable cannot appear in more than one COMMON statement. If you want to pass all variables, use the CHAIN statement with the ALL option and omit the COMMON statement.

Arrays that are passed do not need to be dimensioned in the current program.

Examples:

```
100 COMMON  V,W,X,Y(),Z$
110 CHAIN  "ACCT1"
```

This example chains to program ACCT1 and passes the array Y along with the variables V,W,X, and Z$.

CONT Command

Syntax: **CONT**

Purpose: Continues program execution after a break.

Comments: The CONT command is used to continue program execution after the <Ctrl> <Break> keys are pressed, a STOP or END statement is executed, or an error occurs. Execution resumes at the point where the break occured. In the case of breaks that occur after a prompt from an INPUT statement, execution resumes with the reprinting of the prompt.

The CONT command, when used in conjunction with the STOP statement, is an effective debugging tool. After stopping execution, you can examine or change the values of variables using direct mode statements. Then use CONT to resume execution, or use a direct mode GOTO, which allows you to continue execution at a particular line number.

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

Examples: In this example, a loop is created.

```
100 FOR A=100 to 120
110 PRINT A;
120 NEXT A
RUN
 100 101 102 103 104 105 106 107 108 109 110
```

(at this point we stop the loop by pressing the <Ctrl> <Break> keys.)

```
•
•
•

Break In 120
Ok
CONT
 111 112 113 114 115 116 117 118 119 120
Ok
```

COS Function

Syntax: *v*=**COS**(*x*)

Purpose: Returns the cosine of the range of *x*. The COS(*x*) function is the trigonometric cosine function.

Comments: *x* is the angle whose cosine is to be calculated. The value of *x* is in radians. You can convert degrees to radians by multiplying the degrees by PI/180, where PI=3.141593.

The calculation of COS(*x*) is executed in single precision.

Examples:

```
100 PRINT COS(0)
110 PI=3.141593
120 PRINT COS(PI)
RUN
 1
-1
Ok
```

This example demonstrates that the cosine of 0 radians is equal to 1. Then it calculates the cosine of PI radians.

COMMANDS STATEMENS

CSNG Function

Syntax: *v*=**CSNG**(*x*)

Purpose: Converts *x* to a single-precision number.

Comments: *x* is a numeric expression.

See the CINT function for converting numbers to integer data types.

See the CDBL function for converting numbers to double-precision data types.

Examples:

```
100 K#=123.45678989#
110 PRINT K#; CSNG(K#)
RUN
 123.45678989   123.4568
Ok
```

The value of the double-precision number K# is rounded at the seventh digit.

COMMANDS STATEMENS

CSRLIN Variable

Syntax: *v*=**CSRLIN**

Purpose: Returns the vertical coordinate of the cursor position.

Comments: The CSRLIN variable returns the current line (row) position of the cursor on the active page with the returned value being in the range of 1 to 25.

The POS function provides the column location of the cursor.

The LOCATE statement allows you to set the cursor line.

Examples:

```
100 Y=CSRLIN
110 REM Records the current line
120 X=POS(0)
130 REM Records the current column
140 LOCATE 12,1
150 REM Moves the cursor to line 12, column 1
160 PRINT "WELCOME"
170 LOCATE Y,X
180 REM Restores the cursor to its old position
```

This example stores the cursor coordinates in the variables X and Y, then moves the cursor to line 12, column 1 to display the "WELCOME" message. Then the cursor is restored to its original position.

CVI, CVS, CVD Functions

Syntax: *v*=**CVI**(*2-byte string*)

v=**CVS**(*4-byte string*)

v=**CVD**(*8-byte string*)

Purpose: Converts string values to numeric values.

Comments: Numeric values that are read from a random-access disk file must be converted from strings into numbers. The CVI function converts a two-byte string to an integer. The CVS function converts a four-byte string to a single-precision number. The CVD function converts an eight-byte string to a double-precision number.

The CVI, CVS, CVD functions only change the way BASIC interprets bytes. These functions do not change the bytes of the data itself.

The MKI$, MKS$ and MKD$ functions are the complement functions to CVI, CVS, and CVD.

Examples:

```
100 FIELD #1,4 AS A$, 20 AS B$
110 GET #1
120 N=CVS(A$)
```

This example reads a field from file #1 as defined in line 100. Line 110 reads a record from the file. Line 120 employs the CVS function to interpret the first four bytes (A$) of the record as a single-precision number.

DATA Statement

Syntax: **DATA** *constant*[,*constant*]...

Purpose: Stores (for later access by the program's READ statements) numeric and string constants.

Comments: *constant* may be a numeric or string constant; however, no expressions are allowed. The numeric constants may be in any of the following formats:

integer

fixed point

floating point

hex

octal

String constants in the DATA statements need to be surrounded by quotation marks only if they contain:

commas

colons

significant leading or trailing blanks

DATA statements are not executed and may be included anywhere in the program. They may contain as many constants as a line will hold. Any number of DATA statements can be used in a program. The contents of the DATA statements may be thought of as one continuous list of items, no matter how many items are on a line or where the lines are included in the program. The DATA statements are accessed in line number order by the READ statements.

The variable type (whether it is a numeric or string constant) is specified in the READ statement. The READ statement must agree with the corresponding constant in the DATA statement, otherwise a "Syntax error" occurs.

You may use the RESTORE statement to reread any line in the list of DATA statements.

Examples: Refer to the READ statement for examples of the DATA statement.

DATE$ Statement and Variable

Syntax: When used as a statement:

**DATE$**=*x$*

When used as a variable:

*v$*=**DATE$**

Purpose: Sets or retrieves the current date.

Comments: When used as a statement (DATE$=x$):

x$* is a string expression you may use to set the current date. Enter *x$ in one of the following forms:

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

You must enter a year in the range 1980 to 2099. If you enter a one digit month or day, a 0(zero) is assumed in front of it. If you enter a one digit year, a zero is appended to make it two digits. If you enter a two digit year, 00–77, the year is assumed to be 20*yy*, if you enter 80–99, the year is assumed to be 19*yy*.

When used as a variable (v$=DATES$):

v$ is a 10-character string of the form *mm-dd-yyyy*. *mm* is two digits for the month, *dd* is two digits for the day of the month, and *yyyy* is four digits for the year. The current date may have been set by DOS prior to entering BASIC.

Examples:

```
100 DATE$="7/30/84"
110 PRINT DATE$
RUN
07-30-1984
Ok
```

This example sets the date to July 30th, 1984. Notice a zero was included in front of the month to make it two digits, and the year became 1984. Also notice, that the slashes were replaced by hyphens.

DEF FN Statement

Syntax: **DEF FN***name*[(*arg* [,*arg*]...)]=*expression*

Purpose: Defines and names a function written by you.

Comments: *name* is a legal variable name. This name becomes the name of the function when preceded by FN.

arg is an argument for a variable name in the function definition that will be replaced with a value when the function is called. The arguments in the list represent, in a one-to-one relationship, the values that are to be given in the function call.

expression is an expression for the operation of the function you want to perform. The type of the *expression* (whether it is numeric or string) must equal the type declared by *name*.

The definition of the function is limited to one line. Arguments (*arg*) that are in the function definition serve only to define the function; they do not affect program variables with the same name. A variable name is the *expression* does not have to be in the list of arguments. If it is, the value of the argument is supplied when the function is called. In all other cases, the current value of the variable is used.

The function type determines whether the function is returned as a numeric or string value. The function type is declared by *name*, in the same way as variables are declared. If the type of *expression* (whether it is string or numeric) does not match the function type, you will get a "Type mismatch" error. If the function is numeric, the value of the expression will be converted to the precision specified by *name* before it is returned to the calling statement.

You must define a function with the DEF FN statement before you may call that function. If a function is called before it has been defined, you will get an "Undefined user function" error. However, a function may be defined more than once, in which case, the most recently executed definition is used.

Note: You may have a *recursive* function (one which calls itself). However, you must provide a way to stop the recursion, or you will get an "Out of memory" error.

DEF FN is illegal in direct mode.

Examples:

```
100 PI=3.141593
110 DEF FNA(x)=x/180*PI
120 INPUT "Enter degree", DEGREE
130 PRINT FNA(DEGREE)
```

Line 110 defines the function FNA. This function converts degree into radian. Line 130 calls the function.

COMMANDS STATEMENS

DEF SEG Statement

Syntax: **DEF SEG** [=*address*]

Purpose: Defines the current segment of storage to be referenced by subsequent BLOAD, BSAVE, CALL, PEEK, POKE, or USR. The BLOAD, BSAVE, CALL, PEEK, POKE, or USR definition generates the physical address using the segment specified by DEF SEG and an offset specified by the statement or command above.

Comments: *address* is a numeric expression within the range 0 to 65535.

The beginning setting for the segment when BASIC is started is BASIC's Data Segment (DS). BASIC's DS is the starting point for the user workspace in memory. When you execute a DEF SEG statement that changes the segment, the value does not get reset to BASIC's DS when executing a RUN command.

DEF SEG without the *address* option sets the segment to BASIC's DS.

If the *address* option is included, it should be a value figured on a 16 byte boundary. The value is shifted 4 bits to the left (multiplied by 16) to create the segment address for the subsequent operation. Therefore, if the *address* option is stated in hexadecimal, a 0 (zero) is added to arrive at the actual segment address. BASIC does not perform any checking to validate the segment value.

You must separate with DEF and SEG a space or BASIC will interpret such a statement as DEFSEG=300 to mean: "assign the value 300 to the variable DEFSEG."

Examples:

```
100 REM restores segment to BASIC DS
110 DEF SEG

100 REM sets segment to color screen buffer
110 DEG SEG=&HB800
```

In the second example, the screen buffer is at absolute address B8000 hex. Because segments are specified on 16 byte boundaries, the final hex digit is dropped on the DEF SEG specification.

DEFtype Statements

Syntax: **DEF***type letter*[*-letter*] [*,letter* [*-letter*]]...

Purpose: Declares variable types as one of the following:

integer
single-precision
double-precision
string

Comments: *type* must be INT, SNG, DBL, or STR.

letter is an alphabetic character (A–Z).

A DEF*type* statement declares that the variable names starting with the letter or letters specified in DEF*type* will be the specified type of variable. However, a type declaration character (%, !, # or $) takes precedence over a DEF*type* statement in determining the variable type.

If no type declaration statements are found, BASIC interprets all variables without declaration characters as single-precision variables.

Type declaration statements should appear at the beginning of the program. That way, the DEF*type* statements will always be executed before you use any variables they declare.

Examples:

```
100 DEFDBL  E-I
110 DEFSTR  J
120 DEFINT  Z,K-O
```

Line 100 declares that variables starting with the letters E, F, G, H, or I will be double-precision variables.

Line 110 declares that variables starting with the letter J will be string variables.

Line 120 declares that variables starting with the letters Z, K, L, M, N, or O will be integer variables.

DEF USR Statement

Syntax: **DEF USR**[*n*]=*offset*

Purpose: Specifies the beginning address of a machine language subroutine that will be called later with USR function.

Comments: *n* may be a number from 0 to 9. It declares the number of the USR routine whose address is being specified. If *n* is omitted, BASIC will assume DEF USR0.

offset is an integer expression within the range 0 to 65535. The value of *offset* is added to the value of the current segment to obtain the physical starting address of a machine language subroutine.

You may redefine the address for a USR routine. Any number of DEF USR statements may appear in a program. This allows you to access as many subroutines as necessary. The most recently executed value will be used for the offset.

Examples:

```
100 CLEAR, &HD000
110 DEF USR3=&HD000
120 X=USR3 (Y+2)
```

In this example, the USR3 routine which is loaded at offset D000 hex in BASIC's data segment is called.

DELETE Command

Syntax: **DELETE** [*line1*] [*-line2*]

DELETE [*line1-*]

Purpose: Deletes program lines or line ranges.

Comments: *line1* is the line number of the initial line to be deleted.
line2 is the line number of the final line to be deleted.

The DELETE command removes the specified line(s) from the program. Note that DELETE *line1-* is offered only by BASIC 2.0 and later. BASIC returns to command level after a DELETE is executed.

You may use a period (.) in place of the line number to indicate the current line. If you specify a line number that does not exist in the program, you will get an "Illegal function call" error.

Examples: This example deletes line 110:

```
DELETE 110
```

This example deletes lines 110 through 200, inclusive:

```
DELETE 110–200
```

The following example deletes all lines up to and including line 110:

```
DELETE -110
```

The last example deletes all lines from 110 through the end of the program, inclusive:

```
DELETE 110-
```

DIM Statement

Syntax: **DIM** *variable(subscripts)* [*,variable(subscripts)*]...

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

Comments: *variable* is the name that will be used for the array.

subscripts are numeric expressions separated by commas, which specify the dimensions of the array.

The DIM statement initializes the elements of specified numeric arrays to zero. String array elements are variable length, and initially null, i.e., zero length.

If you use an array variable name without a DIM statement, the maximum value of its subscript is assumed to be 10. If you use a subscript greater than the specified maximum, a "Subscript out of range" error occurs.

The minimum value for a subscript is always 0, unless you specify otherwise with the OPTION BASE statement. You may use a maximum of 255 dimensions for an array with a maximum number of elements per dimension of 32767. Both these numerical limits are also limited by the size of memory and by the length of statements.

If you use DIM with same *variable* more than once, you will get a "Duplicate Definition" error. You may redimension an array by using an ERASE statement before the second DIM.

Examples:

```
10 DIM A(15)
20 FOR I=0 TO 15
30 A(I)=2+I*3
40 NEXT I
```

DRAW Statement

(BASICA only)

Syntax: **DRAW** *string*

Purpose: Draws a figure specified by *string*. Graphics mode only.

Comments: The DRAW statement is used to draw with a *Graphics Macro Language™ (GML™)*. The graphics commands are contained in the *string* expression, which defines an object to be drawn by BASIC. BASIC examines *string* and interprets the single-letter commands it contains. These commands are as follows:

> **NOTE:** Graphics Macro Language™ is a trademark of Microsoft Corporation.

Movement Commands begin movement from the last point referenced (which is the last point drawn by a command).

Command	*Direction of Movement*
U n	Up
D n	Down
L n	Left
R n	Right
E n	Diagonally up and right

F n	Diagonally down and right
G n	Diagonally down and left
H n	Diagonally up and left

In each of the above commands, *n* indicates the distance to move. The number of points moved is *n* times the scaling factor (see the S command below).

M x, y	Move absolute or relative. If *x* is preceded by a plus (+) or minus (−) sign, it is relative. Otherwise, *x* is absolute.

The screen's aspect ratio determines the spacing of horizontal, vertical, and diagonal points. The standard aspect ratio of 4/3 indicates that the screen's horizontal axis is 4/3 as long as the vertical. This information can be used to determine how many vertical points are equal in length to a given number of horizontal points.

For example, medium resolution utilizes 320 horizontal points and 200 vertical points. With the standard aspect ratio of 4/3, in medium resolution 12 horizontal points are equal to 10 vertical points. Thus, to produce a square in medium resolution, the following command would be used:

```
DRAW "U60 R72 D60 L72"
```

Similarly, again with the standard aspect ratio of 4/3, in high resolution 24 horizontal points would be equal to the length of 10 vertical points.

The following Commands may precede any of the Movement Commands.

B	Move, but do not plot points.
N	Move, but then return to the previous position.

The following graphics commands may be used as well:

A n	Set angle *n*, where *n* is a number from zero to three, with zero equal to zero degrees, 1 equal to 90 degrees, 2 equal to 180 degrees, and 3 equal to 270 degrees. A figure that is rotated 90 or 270 degrees will be scaled so that it appears to be the same size as one rotated zero or 180 degrees, with a standard aspect ratio of 4/3.
TA n	Turn angle *n*, where *n* is a number from −360 to +360. When *n* is a positive number, the angle turns counterclockwise. When *n* is negative, the angle turns in a clockwise direction. If you enter a value for *n* that is not within the acceptable range, −360 to +360, the System will display an error message: Illegal function call. This command is supported by BASIC 2.0 and later.

C n	Set color *n*. *n* may range from zero to three in medium resolution, or, in high resolution, be either zero or one. In medium resolution, *n* is used to select a color from current palette defined by the COLOR statement. 0 corresponds to background color, with the default the foreground color, number 3. In high resolution, *n* equal to zero indicates black, and the default (one) specifies white.
S n	Set scale factor. *n* can be any number from one to 255. The scale factor is equal to *n* divided by four. The actual distance moved is derived by multiplying the scale factor by the distances given with the U, D, L, R, E, F, G, H, and relative M commands. The default scale factor is one (n=4).
X variable;	Execute substring. This lets you execute a second string from within the DRAW statement *string*.

P paint, boundary

Set foreground color to *paint* and border color to *boundary*. Both *paint* and *boundary* must be specified, or the System will display an error message. The value of *paint* can be 0, 1, 2, or 3. In medium resolution, the color specified by *paint* is the color from the current palette, defined by the COLOR statement. In high resolution, however, 0 specifies black and 1 specifies white. *Boundary* sets the value of the border color. It must be in the range 0 to 3. This command can be used only in BASIC 2.0 and later. It is not applicable to tile painting.

In all graphics commands, the *n, x,* or *y* argument may either be a constant such as 123, or it can be *=variable;* where *variable* is a numeric variable name. The semicolon (;) is required when a variable is used this way, or in the X command. Otherwise, a semicolon is optional as a command delimiter. Spaces are not significant in *string*.

Variables can also be specified in the form VARPTR$(*variable*), instead of *=variable;*. (This is the only acceptable form in complied programs.)

For example:

One Method:

```
DRAW "XC$;"

DRAW "P=PICTURE;"
```

Another Way:

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

DRAW "P="+VARPTR$(PICTURE)
```

The X command can be very useful, because it lets you define a portion of a figure separately from the rest. For example, a branch as part of a tree. X can also be used to draw a string of commands that is longer than 255 characters.

Full Line Clipping is now performed. In BASIC 1.0, when a line was drawn outside of the Screen it was not properly truncated when it reached the screen's edge. Instead, it was folded back into the screen in a bizarre fashion. With Line Clipping, points plotted outside of the Screen or VIEW port limits do not appear, and lines that intersect the screen or VIEW port limits will appear, cross the screen (or VIEW port) and then disappear correctly at the other end.

Examples: To draw a square box:

```
100 SCREEN 1
110 U$="U30;":D$="D30;":L$="L40;":R$="R40;"
120 BOX$=U$+R$+D$+L$
130 DRAW "XBOX$;"
```

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

To draw a triangle:

```
100 SCREEN 1
200 DRAW "E20 F20 L40"
```

To draw some spokes:

```
100 SCREEN 1 :CLS
110 FOR D=0 TO 360 STEP 5 'Draw some spokes
120 DRAW "TA=D;NU50"
130 NEXT D
```

To draw a painted box:

```
100 SCREEN 1: CLS
110 DRAW "U50R50D50L50"
120 DRAW "BE10"
130 DRAW "P1, 3"
```

EDIT Command

Syntax: **EDIT** *line*

Purpose: Displays a program line to be edited.

Comments: *line* is the line number of a program line. If the program does not contain such a line, you will get an "Undefined line number" error message. A period (.) can be used to indicate the current line.

The EDIT statement is used to display a specified line with the cursor positioned under the first digit of the line number. The line may then be modified as desired.

The LIST command may also be used to display program lines for editing.

END Statement

Syntax: END

Purpose: Terminates execution of the program, closes all files, and returns BASIC to command level.

Comments: An END statement may appear anywhere in the program, though its use is optional. END differs from STOP in two ways:

STOP causes a Break message to be printed.
All files are closed with END.

Examples:

```
800 IF I>1000 THEN END ELSE I=I+1
```

This statement ends the program if I is greater than 1000; otherwise, the program increases I.

ENVIRON Statement

Syntax: **ENVIRON** *x$*

Purpose: Modifies parameters in BASIC's environment string table.

Comments: *x$* is a valid string expression containing the new environment string parameter.

x$ has the format *parm-id=text*, where *parm-id* is the name of the parameter. *parm-id* is separated from *text* by an equal sign. Everything to the left of the first equal sign is read as the name of the parameter. The first character after the equal sign begins the *text*.

text is the new parameter text. If *text* is a null string, or consists only of a single semi-colon, the parameter is removed and the environment string table is compressed.

If *parm-id* does not exist, *x$* is added at the end of the environment string table.

If *parm-id* exists, it is deleted, the environment string table is compressed, and the new *x$* is added at the end of the table.

ENVIRON may be used to change the "PATH" parameter for a child process or to pass parameters to a child by naming a new environmental parameter.

Examples: If PATH is set to "PATH=A:\INVEST" using the DOS command "PATH", ENVIRON statement can change the PATH.

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

The environmental string table now contains PATH=A:\TAX.

The parameter may be appended by using the ENVIRON$ function in conjunction with the ENVIRON statement.

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

The environment string table now contains

```
PATH=A:\TAX;B\YOURS
```

ENVIRON$ Function

Syntax: *v$*=**ENVIRON$** (*parm-id*)
*v$*=**ENVIRON$** (*n*)

Purpose: Reads specified environment string from BASIC's environment string table.

Comments: *parm-id* is a valid string expression containing the parameter to search for.

n is an integer expression returning a value from 1 to 255.

When a string argument is used, ENVIRON$ returns a string which contains the *text* following *parm-id* from the environment string table if *perm-id* is not found, or is followed by no *text* a null string is returned. If a numeric argument is used, ENVIRON$ returns a string "*parm-id =text*" which is the *n*-th string in the environment string table. A null string is also returned if *n*-th string doesn't exist.

Examples: Using the example from the ENVIRON statement, our environmental string table contains the following PATH=A:\TAX; B:\YOURS.

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

prints the string A:\TAX; B: \YOURS

When DOS starts up, the string "COMSPEC=parameter" is always placed in the environment table. So the PATH is the second string.

```
PRINT ENVIRON$ (2)
```

prints the string as PATH=A:\TAX; B:\YOURS

EOF Function

Syntax: *v*=**EOF** (*filenum*)

Purpose: Designates an end of file condition.

Comments: The EOF function can be used to avoid an "Input past end" error. EOF returns -1 (true) if the end of the specified file has been reached. The function returns zero if the end of the file has not yet been reached.

For BASIC release 2.0 and later, EOF(0) returns the end of file condition when you use standard input devices with redirection of I/O.

EOF is significant only when applied to a file that has been opened for sequential input from disk, or for a communications file. (−1 for a communications file means that the buffer is empty.)

Example:

```
100 OPEN "FILE" FOR INPUT AS #1
200 IF EOF(1) THEN END
300 INPUT #1, D
400 PRINT D:GOTO 200
```

This example reads information from the sequential file named "FILE". Values are read into D until the end of "FILE" is reached.

ERASE Statement

Syntax: **ERASE** *arrayname*[,*arrayname*]...

Purpose: Deletes arrays from a program.

Comments: *arrayname* is the name of the array to be erased.

After arrays are erased, the memory space allocated for them may be used for other purposes.

ERASE can also be used for redimension of arrays. Redimension of an array that has not first been erased causes "Duplicate Definition" error.

ERASE is only applied to *array variables;* CLEAR is used to erase *all* variables from the work area.

Examples:

```
100 BEGIN=FRE(0)
200 DIM ARRAY(100,100)
300 LATER=FRE(0)
400 ERASE ARRAY
500 DIM ARRAY(10,10)
600 LAST=FRE(0)
700 PRINT BEGIN, LATER, LAST
RUN
 49569        8740        49050
Ok
```

Here the FRE function to show how ERASE can be used to free memory. The array ARRAY used about 40K-bytes of memory (49560-8740) when dimensioned as ARRAY(100,100). After it was erased, it could be redimensioned to ARRAY(10,10), and it only required a little more than 500 bytes (49569-49050).

The actual values returned by the FRE function may vary on your computer.

ERDEV and ERDEV$ Variables

Syntax: *v*=**ERDEV**

*v$*=**ERDEV$**

Purpose: Contains the error code and the name of the device causing the error.

Comments: The value of the variable ERDEV is the error code for the latest error. The error code is held in the lower 8 bits, the upper 8 bits contain the word attributes from the device header block.

When the error is on a character device, ERDEV$ contains the device name. If the error is not on a character device, ERDEV$ contains the block device name (A:, B: etc).

Example: While a printer is out of paper, the LLIST command causes a "Device Timeout" error.

ERDEV contains:

2 (device header word attributes)

ERDEV$ contains:

LPT1:

ERR and ERL Variables

Syntax: *v* = **ERR**

v = **ERL**

Purpose: Contains the error code and line number.

Comments: The value of the variable ERR is the error code for the latest error; the variable ERL contains the line number where the error was detected.

These variables are usually used in IF...THEN statements to determine program flow. If you test ERL in this way, be certain to place the line number to the right of the relational operator:

IF ERL=*linenumber* THEN...

The line number must be on the right side of the IF operator. RENUM command interprets it as a line number only when it is on the right side.

If a direct mode statement caused the error, ERL will contain 65535. To avoid renumbering it, use the form:

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

ERR and ERL can be set with the ERROR statement.

BASIC error codes are listed in "Appendix A: Error Messages."

Examples:

```
100 ON ERROR GOTO 400
200 LPRINT "Will the printer print?"
300 END
400 IF ERR=24 THEN LOCATE 23,1:
    PRINT "Check out printer": RESUME
500 PRINT "ERR=";ERR,"ERL=";ERL:END
```

This example tests whether the printer has been loaded with paper, and if it has been powered on.

COMMANDS STATEMENS

ERROR Statement

Syntax: **ERROR** *n*

Purpose: Simulates a BASIC error condition, or lets you define your own error codes.

Comments: *n* is an integer expression from zero through 255.

If the value of *n* equals that of a BASIC error code, the ERROR statement simulates that error's occurrence. If an error handling routine has been defined with the ON ERROR statement, that routine is called. Otherwise, the BASIC error message corresponding to the code is displayed, and execution stops.

To define an error code, number it with a value different from any used by BASIC. (BASIC errors are coded with numbers 1 to 76, consecutive.)

If you define your own error in this way, and don't handle it in a corresponding error handling routine, BASIC will display the message "Unprintable error," and halts execution.

Examples: The first example simulates a "String too long" error.

```
100 T = 15
200 ERROR T
RUN
String too long in 200
```

EXP Function

Syntax: v =**EXP**(x)

Purpose: Computes the exponential function.

Comments: x can be any numeric expression.

EXP returns the number e (base of natural logarithms) raised to the x power. An overflow will occur if x is more than 88.02969.

Examples:

```
100 X = 0
200 PRINT EXP(X+1)
RUN
 2.718282
Ok
```

This example calculates e.

COMMANDS STATEMENS

FIELD Statement

Syntax: **FIELD** [#]*filenum*, *width* AS *stringvar* [,*width* AS *stringvar*]...

Purpose: Defines variables that will be used to access a random file buffer.

Comments: *filenum* is the number associated with the file when it was OPENed.

width is a numeric expression that specifies the number of character positions to be allocated for *stringvar*.

stringvar is a string variable name that will be used to access the random file field.

A FIELD statement defines variables that will be referenced by GET and PUT statements.

For example,

```
FIELD 1, 20 AS NAME$, 10 AS NUM$, 40 AS ADDR$
```

allocates the first 20 positions (bytes) in the buffer of File #1 to the string variable NAME$, the next 10 bytes to NUM$, and the next 40 positions to ADDR$.

FIELD does *not* actually place data in the buffer, nor is it used to manipulate data within the buffer. Placement and manipulation of data is done with the LSET, RSET, GET, and PUT statements.

The total number of bytes allocated with a FIELD statement must be less than the record length specified when the file was OPENed, or, a "Field overflow" error occurs.

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

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

Examples:

```
100 OPEN "CUSTOMER" AS #1
200 FIELD 1, 2 AS NUM$, 20 AS NAME$, 40 AS ADDRESS$
300 LSET NAME$="ATKINS INC"
400 LSET ADDRESS$="2001 MARKET ST, SAN FRANCISCO"
500 LSET NUM$=MKI$(1)
600 PUT 1,1
700 GET 1,1
800 CUSNUM%= CVI(NUM$): CUSNAM$ = NAME$
900 PRINT CUSNUM%, CUSNAM$, ADDRESS$
```

This example opens a file named "CUSTOMER" as a random file. The variable NUM$ is assigned the first 2 positions in each record, NAME$ is assigned the next 20 positions, and ADDRESS$ is allocated the next 40 positions. Lines 300 through 500 place information in the buffer, and the PUT statement in line 600 writes the buffer contents to the file. Line 700 reads the record just written, and line 900 displays the three fields. Note in line 800 that no problem arises when variable name defined in a FIELD statement appears on the *right* side of an assignment statement.

FILES Command

Syntax: **FILES** [*filespec*]

Purpose: Displays the names of files stored on the disk. This command is similar to the DIR command in DOS.

Comments: *filespec* is a string expression that identifies a file specification. When *filespec* is omitted, all files on the current directory will be listed.

All matching filenames will be displayed. The filename may include question marks; these will match with any single character in the filename or extension. An asterisk (*) will match one or more characters starting at that position.

When a drive is designated as part of *filespec*, files matching the specified filename on the current directory of that drive will be listed. If no drive is specified, the default drive is used.

Examples:

```
FILES
```

The names of all files on the default drive will be listed.

```
FILES "*.DAT"
```

The names of all files with an extension of .DAT on the default drive will be listed.

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

The names of all files on drive B will be listed.

```
FILES "NAME??.DAT"
```

This lists each file on the default drive with a filename starting with NAME followed by up to two other characters, and an extension of .DAT.

An enhancement in BASIC 2.0 and later now makes it possible to list all the files in the current directory of drive B by entering:

```
FILES "B:"
```

BASIC not only lists all files in response to this command but also the directory name and the remaining number of free bytes.

When you build stratified directories in BASIC 2.0 and later, you will find that each sub-directory contains two additional entries. These are listed when the FILES command is used to list a sub-directory. The first entry consists of a single period followed by <DIR>, and designates this file as a sub-directory. The second entry consists of two periods followed by <DIR>. This entry serves to find the higher level, or parent, directory in which this sub-directory is contained. For example:

```
FILES "B:\INVEST\BONDS"

. <DIR>                  . . <DIR>

31446 Bytes free
```

The above command causes BASIC to list the files in the files in the current sub-directory, BONDS on drive B. Since BONDS contains no files, 31446 bytes of space are available.

```
FILES "INSURE\"
```

This command lists all files in the sub-directory INSURE, which in this example is a sub-directory of the current directory. If INSURE contains its own sub-directories, these will be labeled <DIR> in the file list.

COMMANDS STATEMENS

FIX Function

Syntax: v = **FIX**(x)

Purpose: Truncates a given x to an integer.

Comments: x can be any numeric expression.

FIX deletes all digits to the right of the decimal point and returns the value of the digits to the left of it.

The difference between FIX and INT is that INT returns the next lower number when x is negative.

Examples:

```
PRINT FIX(21.75)
 21
Ok
PRINT FIX(−5.7)
−5
Ok
```

COMMANDS STATEMENS

FOR and NEXT Statements

Syntax: **FOR** *variable=x* TO *y*[STEP *z*]

•

•

•

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

Purpose: Executes a series of instructions a given number of times.

Comments: *variable* is an integer or single-precision variable that is used as a counter.

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

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

z is a numeric expression that will be used as a counting increment.

The program lines after the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the STEP value (*z*). If *z* is not specified, the increment is assumed to be 1 (one). A check is performed to determine whether the value of the counter is now greater than the final value *y*. If not, BASIC branches back to the statement following FOR and the process is repeated. If the counter has exceeded *y*, execution continues with the statement that follows NEXT. This process is called a FOR...NEXT loop.

If *z* is negative, the incrementing process is reversed. The counter is decremented each time the loop is performed, and execution continues until the counter is less than the final value.

The body of the loop will be skipped if *x* is already greater than *y* when the STEP value is positive, or if *x* is less than *y* when *z* is negative. If z is zero, an infinite loop will be created unless you provide some way to increment the counter.

If you use integer variable as a counter, performance of a program will be improved.

Nested Loops

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

NEXT *var1, var2, var3* ...

is the same as this sequence of statements:

NEXT *var1*
NEXT *var2*
NEXT *var3*
•
•
•

The variable(*s*) may be omitted from the NEXT statement, in which case the statement will be matched with the most recent FOR. And, in fact, omitting the variable name(s) will cause your program to execute somewhat more quickly. But if you are using nested loops, you should include the variable(s) on all the NEXT statements.

If a NEXT statement is placed before its corresponding FOR statement, you will get a "NEXT without FOR" error message.

Examples: The first example shows a FOR...NEXT loop with a counter increment of 2.

```
100 J=8: K=50
200 FOR I=1 TO J STEP 2
300 PRINT I;
400 K=K+5
500 PRINT K
600 NEXT
RUN
 1  55
 3  60
 5  65
 7  70
Ok
```

In the next example, the loop is executed ten times. This is because the final value for the counter variable is always set *before* the initial value is set.

```
100 M=5
200 FOR M=1 TO M+5
300 PRINT M;
400 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10
Ok
```

FRE Function

Syntax: *v*=**FRE**(*arg*)
v=**FRE**(*arg$*)

Purpose: Returns the number of bytes in memory that are not currently in use. This number does not include the reserved portion of the interpreter workarea (approximately 4K bytes).

Comments: *arg* and *arg$* are dummy arguments.

Strings in BASIC are manipulated dynamically because their lengths are variable. Thus, string space may become fragmented.

FRE with any string argument causes a "garbage collection" before returning the number of free bytes. Garbage collection is when BASIC collects all its useful data and frees unused memory areas once used for strings. The data is compressed so that you can continue until you actually run out of space.

BASIC automatically performs a garbage collection when it is running out of usable workarea. You can enter FRE(" ") periodically to achieve shorter delays for each garbage collection.

CLEAR *,n* sets the maximum size of the BASIC workspace. FRE returns the amount of free storage in this workspace. If the workspace is empty, then the value returned by FRE will be approximately 4K bytes (the size of the interpreter workarea) smaller than the number of bytes set by CLEAR.

Examples:

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

The value returned by FRE on your computer may be different from that shown in this example.

GET Statement (Files)

Syntax: **GET** [#]*filenum* [, *number*]

Purpose: Reads a random file record into a random buffer.

Comments: *filenum* is the number associated with the file when it was OPENed.

number is the number of the desired record, and may range from 1 to 32767. If *number* is omitted, BASIC reads the next record (after the last GET).

Once a GET statement has been executed, characters may be read from the buffer via the INPUT # and LINE INPUT # statements, or by references to variables defined in the FIELD statement.

GET can also read data from communication files, in which case number specifies the number of bytes to read from the buffer. This number may not exceed the value set by the LEN option on the OPEN"COM... statement.

Examples: The following example opens the file "NAMES" for random access, with fields defined in line 200. The GET statement on line 300 reads a record into the file buffer on line 400, information from the record is displayed.

```
100 OPEN "NAMES" AS #1
200 FIELD #1,20 AS FNAME$, 20 AS LNAME$,2 AS AGE$
300 GET #1
400 PRINT FNAME$, LNAME$, CVI(AGE$)
```

GET Statement (Graphics)

(BASICA only)

Syntax: **GET** *(x1,y1)–(x2,y2),arrayname*

Purpose: Reads points from a specified screen area and stores them in array form.

Comments: *(x1,y1)* and *(x2,y2)* represent coordinates in either absolute or relative form of two points that are positioned at opposite corners of a rectangle on the screen.

arrayname is the name of the numeric array that will hold the information.

GET reads the colors of the points within the rectangle specified by *(x1,y1)* and *(x2,y2)* and places them into the specified array. The graphics PUT statement can then be used to reference the array and redisplay the image.

The array is used to "file" the image; though it must be numeric, it may be any precision. The screen information is placed in the array as follows:

> the first four bytes contain, respectively, the x and y dimensions in bits.
>
> The succeeding bytes contain the color of each point within the specified area of the screen.

The required size of the total array, in bytes, is computed as follows:

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

where x and y are the lengths of the rectangle's horizontal and vertical sides, and "bpp" is short for "bits per pixel", and equal to 2 in medium resolution, and 1 in high resolution.

Integer array elements are two bytes long, single-precision elements use four bytes each, and each double-precision element is eight bytes long.

The data for each row of points is aligned on an internal byte boundary, so if less than a multiple of eight bits is stored, the remainder is filled with zeros.

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 the first element of the array, and the y dimension is in second element.

GOSUB and RETURN Statements

Syntax: **GOSUB** *line*

•

•

•

RETURN

Purpose: Executes a subroutine and returns to the branch point.

Comments: *line* is the line number of the subroutine's first line.

A subroutine may be called any number of times during a program. A subroutine may also be called from within another subroutine; such subroutine nesting is limited only by available memory.

The RETURN statement in a subroutine causes BASIC to branch back to the statement following the latest GOSUB statement. More than one RETURN statement may be included in a single subroutine, if you want to return from different points in the routine.

To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

To branch to different subroutines depending on the result of an expression, use ON...GOSUB.

Examples:

```
100 GOSUB 400
200 PRINT "SUBROUTINE COMPLETE "
300 END
400 PRINT "EXECUTING ";
500 PRINT "SUBROUTINE"
600 RETURN
RUN
EXECUTING SUBROUTINE
SUBROUTINE COMPLETE
Ok
```

The GOSUB statement on line 100 calls the subroutine beginning on line 400. The program branches to line 400 and begins executing statements 400, 500 and 600. Since RETURN is found on line 600, the program goes back to the statement, line 200, after the subroutine call. The END statement on line 300 prevents the subroutine from being performed again.

GOTO Statement

Syntax: **GOTO** *line*

Purpose: Branches unconditionally out of the current program sequence to a specified line number.

Comments: *line* is a valid line number within the program.

If *line* is the line number of an executable statement, that statement and those following will be executed. If *line* is a non-executable statement (such as REM or DATA), execution proceeds at the first executable statement encountered after *line*.

In direct mode you can use GOTO to re-enter a program at a desired point. It can be helpful when debugging a program.

To branch to different points depending on the result of an expression, use ON...GOTO.

Examples:

```
50  DATA 2,10,15
100 READ RAD
200 PRINT "RAD =";RAD,
300 AR = 3.14*RAD^2
400 PRINT "AREA =";AR
500 GOTO 50
RUN
RAD= 2      AREA = 12.56
RAD=10      AREA = 314
RAD=15      AREA = 706.5
Out of DATA in 100
Ok
```

GOTO on line 500 could cause an infinite loop, which is stopped when the program runs out of data in the DATA statement, since the DATA statement has only 3 data.

HEX$ Function

Syntax: *v$*=**HEX$**(*n*)

Purpose: Converts a decimal argument to a hexadecimal value represented by a string.

Comments: *n* is a numeric expression ranging from −32768 to 65535 and rounded to an integer before HEX$ is evaluated.

The two's complement form of a negative *n* is used. That is, HEX$(−*n*) is identical to HEX$ (65536−*n*).

Refer to the OCT$ function for octal conversion.

Examples: In the following example the HEX$ function computes the hexadecimal euivalent of the two decimal values entered.

```
10 INPUT D
20 H$ = HEX$ (D)
30 PRINT D "DECIMAL IS EQUAL TO " H$ " HEXADECIMAL"
RUN
?  32
  32 DECIMAL IS EQUAL TO 20 HEXADECIMAL
Ok
RUN
? 1023
  1023 DECIMAL IS EQUAL TO 3FF HEXADECIMAL
Ok
```

IF Statement

Syntax: **IF** *expression* [,]THEN *clause* [ELSE *clause*]

IF *expression* [,]GOTO *line* [[,] ELSE *clause*]

Purpose: Used to make a decision regarding program flow depending on the result of an expression.

Comments: *expression* can be any numeric expression.

clause may be either a single statement or a sequence separated by colons, or it may simply be a line number to which the program might branch.

line is the line number of an existing program line.

If the result of *expression* is true (not zero), BASIC executes the THEN or GOTO clause. The THEN clause may include either a line number for branching or one or more executable statement. GOTO must always be followed by a line number.

If the *expression* is false (zero), the THEN or GOTO clause is ignored and the ELSE clause, if included, is executed. Execution then continues with the next executable statement following the IF statement.

Testing Equality: when testing equality of a value resulting from a single- or double-precision computation, remember that the internal representation of the value may not be exact, because single- and double-precision values are stored in floating point binary format. Therefore, you should test instead against the *range* over which the accuracy may vary. For example, test a computed variable X against the value 1.0 this way:

```
IF ABS (X−1.0)<1.0E−6 THEN ...
```

A true result will be returned when the value of X is 1.0 with a relative error of under 1.0E−6.

Nested IF Statements: IF...THEN...ELSE statements may be nested. Nesting of IF... THEN... ELSE statements is limited only by the line length. For instance,

```
IF A>B THEN PRINT "MORE" ELSE IF B>A
   THEN PRINT "LESS" ELSE PRINT "THE SAME"
```

is a valid statement. A statement need not contain a corresponding number of ELSE and THEN clauses; each ELSE is matched with the closest unmatched THEN, as shown by the following example:

```
IF X=Y THEN IF Y=Z THEN PRINT "X=Z"
   ELSE PRINT "X<>Z"
```

"X<>Z" will not be printed if X<> Y, that is, if (X=Y) is false.

Examples: The following statement GET record J if J is not zero:

```
20 IF J THEN GET #1,J
```

In the following example, if J is greater than 10 and less than 20, NUM is calculated and the program branches to line 30. If J is not in this range, the message "BEYOND RANGE" is printed. Note that two statements comprise the THEN clause.

```
10 IF (J>10) AND (J<20) THEN NUM=1984-1: GOTO 30 ELSE PRINT
"BEYOND RANGE"
```

The next example statement causes printed output to go to either the screen or the printer, depending on the value of a variable named FLAG. If FLAG is false (zero), output goes to the printer; otherwise, output will be printed on the screen:

```
200 IF FLAG THEN PRINT N$ ELSE LPRINT N$
```

INKEY$ Variable

Syntax: *v*$=**INKEY$**

Purpose: Reads a character entered from the keyboard.

Comments: INKEY$ reads one character, even if the keyboard buffer contains several. The returned value will be in the form of either a zero-, one-, or two-character string as shown below.

> A null string (zero length) indicates that no character is waiting at the keyboard to be read.
>
> A one-character string contains the character read from the keyboard buffer.
>
> A two-character string contains a special extended code, the first character of which will be hex zero.

The value of INKEY$ must be assigned to a string variable before the character can be utilized with any BASIC statement or function.

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

> Ctrl-Break, which halts the program
> Ctrl-Num Lock, which makes the system pause
> Alt-Ctrl-Del, which performs a System Reset
> PrtSc, which prints the screen content

Pressing <ENTER> key in response to INKEY$ passes the carriage return character to the program.

Examples: In the following examples, if you press <Y> key, the execution is continued. <N> key ends this program.

```
100 PRINT "Continue? (Y/N)"
110 A$=INKEY$
120 IF A$="Y" THEN GOTO 150
130 IF A$="N" THEN END
140 GOTO 110
150 • • •
```

INP Function

Syntax: *v*=**INP**(*n*)

Purpose: Returns a byte read from input port *n*.

Comments: *n* must be within the range of zero to 65535.

INP performs a complementary function to the OUT statement, the same function as the assembly language IN instruction.

Examples:

```
100 X=INP(316)
```

This line reads a byte from port 316 and assigns it to the variable X.

INPUT Statement

Syntax: **INPUT**[;]["*prompt*";] *variable*[,*variable*]...

Purpose: Receives data from the keyboard during program execution and assigns each item to a specified variable.

Comments: "*prompt*" is a string constant used to prompt for the input.

variable is the numeric, string variable or array element names to which the entered data will be assigned.

When an INPUT statement occurs, the program displays a question mark on the screen to indicate the program is waiting for data. If a "*prompt*" is included, the string precedes the displayed question mark. The required data is then entered at the keyboard.

A comma rather than a semicolon may be used after the prompt string to suppress the question mark. For example, the statement

```
INPUT "ENTER TODAY'S DATE", D$
```

prints the prompt without the question mark.

Each entered data item's type must agree with that specified by the variable name. (Strings entered in response to an INPUT statement need only be surrounded by quotation marks if they contain commas or significant leading or trailing blanks.)

Entering too many or too few items, or an incorrect type of value (numeric instead of string, etc.) causes the message "?Redo from start".

If a semicolon is placed immediately after the word INPUT, then pressing Enter to input data will not cause a carriage return/line feed sequence on the screen. This means that the cursor remains in the same position as when you pressed Enter.

Examples:

```
100 INPUT A
200 PRINT A "SQUARED IS" A^2
300 END
RUN
?
```

The question mark indicates that the computer wants you to enter something. Suppose you enter a 3. The following will be displayed:

```
? 3
 3 SQUARED IS 9
Ok
```

```
100 PI=3.14
200 INPUT "ENTER THE RADIUS";RAD
300 AREA=PI*RAD^2
400 PRINT "THE CIRCLE'S AREA IS";AREA
500 END
RUN
ENTER THE RADIUS?
```

For this example, the prompt included in line 200 causes the computer to prompt with "ENTER THE RADIUS?" Suppose you enter 6. Execution continues:

```
ENTER THE RADIUS? 6
THE CIRCLE'S AREA IS 113.04
Ok
```

INPUT # Statement

Syntax: **INPUT** *#filenum, variable* [*,variable*]...

Purpose: Reads data items from a device or file and assigns each to a program variable.

Comments: *filenum* is the number assigned when the file was OPENed.

variable is a variable name that will be assigned to the items in the file. It can be a string or numeric variable, or an array element.

The file may be a sequential or random disk file, a sequential data stream from a serial interface, or the keyboard (KYBD:). The type of data in the file must be the same as that specified by the variable name.

The data items in the file should appear as they would if the data were being entered 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 line feed is assumed to be the beginning of the number. The number is considered to end if a space, carriage return, line feed, or comma is encountered.

BASIC scans the data for a string item in the same way. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of the string item. If this first character is a double quote ("), the string item will consist of all characters read between the first double quote and the next. Thus, a string enclosed in quotes may not contain a double quote as a character. If the first character of the string is not a double quote, it will end when a comma, carriage return, or line feed is encountered, or after 255 characters have been read. If the end of the file is reached while a numeric or string item is being INPUT, the item is cancelled.

INPUT$ Function

Syntax: *v$*=**INPUT$** (*n*[,[#]*filenum*])

Purpose: Returns a string of *n* characters, read from the keyboard or from a specified file.

Comments: *n* is the number of characters to be read.

filenum is the number assigned when the file was OPENed. No *filenum* means reading from the keyboard.

When the keyboard is used for input, no characters are displayed on the screen. All characters (including control characters) are passed except Ctrl-Break, which is used to interrupt the INPUT$ function. When responding to INPUT$ from the keyboard, you need not press Enter.

The INPUT$ function lets you enter characters from the keyboard that are significant to the BASIC program editor, such as Backspace (ASCII code 8). INPUT$ or INKEY$ (*not* INPUT or LINE
INPUT) should be used to read in these special characters. Similarly, with communications files, the INPUT$ function is preferred over the INPUT # and LINE INPUT # statements, since any ASCII character may be significant in communications.

Examples: This example program lists the contents of a sequential file in hexadecimal.

COMMANDS STATEMENS

```
100 OPEN "I",1,"DATA"
200 IF EOF (1) THEN 500
300 PRINT HEX$ (ASC(INPUT$(1,#1)));
400 GOTO 200
500 PRINT
600 END
```

The next example will read a single character from the keyboard entered in response to a displayed question.

```
10 PRINT "TYPE G TO GO OR S TO STOP"
20 CODE$=INPUT$(1)
30 IF CODE$="G" THEN 50
40 IF CODE$="S" THEN 70 ELSE 10
```

INSTR Function

Syntax: *v*=**INSTR**([*n*,]*x$*,*y$*)

Purpose: Searches for the first occurrence of string *v$* in string *x$* and returns the position at which it was found. The optional offset *n* specifies the position in *x$* where the search should begin.

Comments: *n* is a numeric expression that may range from 1 to 255.

x$ and y$ can be string variables, expressions, or constants.

INSTR returns zero if:

n is greater than the length of x$, or
x$ is a null (zero length) string, or
y$ cannot be located

INSTR returns *n* (or 1 if *n* is not given) if *y$* is null.
If *n* is out of range, BASIC will display an "Illegal function call" message.

Examples: The following example searches for "B" in the string "ABCDEB". First the string is searched from the beginning and "B" is found at position 2; then the search starts at position 4, and "B" is first located at position 6.

```
10 X$="ABCDEB"
20 Y$="B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

COMMANDS STATEMENS

INT Function

Syntax: *v*=**INT**(*x*)

Purpose: Returns the largest integer that is less than or equal to *x*.

Comments: *x* is any numeric expression.

See the FIX and CINT functions, which also return integer values.

Examples:

```
PRINT INT (99.89)
  99
Ok
PRINT INT (−12.11)
−13
Ok
```

COMMANDS STATEMENS

KEY Statement

Syntax: **KEY** *n, x$*

KEY ON

KEY OFF

KEY LIST

Purpose: Assigns automatic functions to keyboard function keys, and references and manipulates the settings.

Comments: *n* is the number of a keyboard function key, an integer in the range from 1 to 10. For BASIC 2.0, *n* can be in the range from 1 to 10 and 15 to 20.

The **KEY** *n, x$* statement allows function keys to be designated for special " soft key" functions, in case *n* is in the range from 1 to 10.

The soft keys are initially assigned as follows:

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

The "←" means the <ENTER> key.

KEY ON displays the soft key assignments on screen line 25. If the screen width is 40, five of the ten keys will be displayed. If the width is 80, all ten will be displayed. In either case, only the first six characters of each value are displayed. KEY ON is the default for the soft key display.

KEY OFF erases the soft key assignment display from line 25, making that line available for program use. However it does not deactivate the function keys.

Once the soft key display is turned off, use LOCATE 25, 1 followed by PRINT to display anything on the last screen line. Unlike lines 1-24, line 25 is not scrolled.

KEY LIST causes all ten soft key values to be listed on the screen in their entirety.

Assigning a null (zero length) string to a soft key disables the function assigned to that key. This can be done as follows:

```
150 KEY 10, ""
```

When a soft key is pressed, the INKEY$ function returns one character of the assigned string for each call. If the soft key is disabled, INKEY$ will return a two-character string. The first character will be zero, and the second ASCII key scan code.

Advanced BASIC 2.0 provides for six more key traps. These enable you to trap Ctrl, Shift, and super-shift keys. The syntax for defining these keys is:

KEY *n*, CHR$ (*s*)+CHR$(*t*)

IN the above statement, *n* is an integer from 15 to 20, and *s* is a number corresponding to the hex values of keys to be shifted. Latched key hex values are:

<CAPS LOCK> &H40
Caps Lock is active

<NUM LOCK> &H20
Num Lock is active

<ALT> &H08
Alt key is depressed

<Ctrl> &H04
Ctrl key is depressed

<Shift> &H01, &H02, &H03
One or both Shift keys are depressed with the same result.

Hex values for key shift states can be added together. Thus, the Ctrl and Alt keys could be added together.

t is an integer designating one of the 83 physical keys on the keyboard. See APPENDIX C for a table of the integers associated with each key.

BASIC processes trapped keys in the following order:

1. The line printer key, <Ctrl> <PrtSc>, is processed first. Note that even when <Ctrl> <PrtSc> is defined as a key trap, it can still be used to get a printed copy of screen display.

2. User-defined keys 15 to 20 are processed.

3. Finally, function keys and cursor direction keys are processed. Defining any of these keys as trappable will have no effect as they are considered pre-defined.

   NOTE: Trapped keys are not passed on the keyboard buffer. They are not read by BASIC.

   Exercise caution when trapping Ctrl-Break and Ctrl-Alt-Del. Unless you have included a test in your trap routine, the Sr. Partner will have to be powered off to stop the program.

Examples:

```
100 KEY 16, CHR$(&H40)+CHR$(34)
110 ON KEY (16) GOSUB 1000
120 KEY (16) ON
```

The above example sets a key trap for capital C. Note that key trapping uses all KEY statements, including KEY, KEY(*n*), and ON KEY.

```
200 KEY 18, CHR$(&H04+&H08)+CHR$(37)
```

This sets a key trap for <Ctrl> <Alt> K. Note that values for <Ctrl> and <Alt> were summed.

KEY(n) Statement

(BASICA only)

Syntax: **KEY**(*n*) ON

KEY(*n*) OFF

KEY(*n*) STOP

Purpose: Enables and disables trapping of a specified key in a BASIC program.

Comments: *n* represents a numeric expression whose value ranges from one to 20, and specifies the key to be trapped:

1-10	function keys F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-20	keys defined by the syntax: KEYn,CHR$(*s*)+CHR$(*t*) (keys 15-20 are trappable only in BASIC 2.0 and later)

KEY(*n*) ON enables trapping of function key or cursor control key activity. After KEY(*n*) ON statement, if a non-zero line number was included in the ON KEY(*n*) statement then whenever BASIC starts a new statement it checks to see if the specified key was pressed. If so, it will perform a GOSUB to the specified line number.

KEY(*n*) OFF disables trapping; if the key is pressed, the event is not noted.

After KEY(*n*) STOP, no trapping takes place, but if the specified key is pressed, the action is remembered, and an ON KEY statement will be executed as soon as trapping is enabled.

KILL Command

Syntax: **KILL** *filespec*

Purpose: Erases a disk file.

Comments: *filespec* is a valid file specification as defined in Chapter 5. For BASIC 2.0 and later, the specification can include a pathname.

If the file extension exists, the filename must include it. For instance, you can save a BASIC program with the command

```
SAVE "ACCT"
```

and BASIC will supply the extension .BAS. If you later want to delete that program, however, you must use KILL "ACCT.BAS", rather than KILL "ACCT".

If a KILL command is entered for a currently-open file, a "File already open" error messge will be displayed.

> **NOTE:** KILL is used only to erase files. You must use the RMDIR command to delete directories.

Examples: To delete a file named "DATA", you could use:

```
500 KILL "DATA"
```

To erase a file named "IRA" in the TAX subdirectory, you might enter:

```
KILL "INVEST\TAX\IRA"
```

COMMANDS STATEMENS

LEFT$ Function

Syntax: *v$*=**LEFT$** (*x$*,*n*)

Purpose: Returns a string comprising the leftmost *n* characters of *x$*.

Comments: *x$* may be any string expression.

n is a numeric expression in the range of zero to 255 which specifies the number of characters to be returned.

If *n* is greater than the length of the target string *x$*, the entire string is returned. If n is zero, a null string (zero length) is returned.

Also see the MID$ and RIGHT$ functions.

Examples: In the example below, the LEFT$ function is used to extract the first three characters from the string "ACCOUNTING PROGRAM".

```
100 A$="ACCOUNTING PROGRAM"
200 B$=LEFT$(A$,3)
300 PRINT B$
RUN
ACC
Ok
```

LEN Function

Syntax: *v*=**LEN**(*x$*)

Purpose: Returns the number of characters in *x$*.

Comments: *x$* may be any string expression.

Unprintable characters and blanks are included in the total number of characters.

Examples:

```
100 x$= "SAN FRANCISCO, CA"
200 PRINT LEN (x$)
RUN
 17
Ok
```

There are 17 characters in the string "SAN FRANCISCO, CA", which includes a comma and two blanks.

LET Statement

Syntax: **[LET]** *variable=expression*

Purpose: Assigns the value of an expression to a variable.

Comments: *variable* is the name of a string or numeric variable or array element that is to be assigned a value.

expression represents the value that will be assigned to *variable*. The expression's type must match the variable's type, or "Type mismatch" message will be displayed.

The word LET is optional, that is, the equal sign is sufficient for assigning an expression to a variable name.

Examples:

```
100 LET NUM=10
200 LET E=NUM+5
300 LET PROG$="ACCOUNTING"
```

This example assigns the value 10 to the variable NUM. It then assigns the value 15 (the value of the expression NUM+5) to the variable E. The string "ACCOUNTING" is assigned to the variable PROG$.

The same result could have been achieved as follows:

```
100 NUM=10
200 E=NUM+5
300 PROG$="ACCOUNTING"
```

LINE Statement

Syntax: **LINE** [(*x1,y1*)]−(*x2,y2*) [,[*color*] [,B[F]]] [,*style*]]

Purpose: Draws a line or a box on the screen. Graphics mode only.

Comments: (*x1,y1*) is the cordinate for the starting point of the line.

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

color is a number that may range from zero to 3. In medium resolution, the color is selected from the current palette as defined by the COLOR statement. In high resolution, if *color* is specified as zero, the background color (black) is used. If *color* is omitted, the default is used: white, the foreground color, which is color number one.

B is included to specify the drawing of boxes on the screen. B is used to draw a rectangle with points (*x1,y1*) and (*x2,y2*) as opposing corners. Four LINE would be needed to do the same thing, to draw four lines connecting four different points.

BF has the same effect as B, but in addition, all the interior points are also displayed in a selected (or default) color.

style is a 16-bit integer mask used to put pixels on the screen. Known as line styling, the mask is supported only in BASIC 2.0 and later.

To store a pixel on the screen, LINE uses the current circulating bit in *style*. If the bit is zero, no point will be stored. If the bit is one, a point will be plotted. After each point, the next bit position in *style* is selected.

style is used for normal lines and boxes, but has no effect on filled boxes. Using the *style* option with BF will cause BASIC to display a Syntax error.

Because a zero bit skips over a point on the screen without erasing it, you may wish to specify a background line before the styled line. This technique forces a known background.

Use *style* to draw a dotted line by storing every other point. The pattern for a dotted line will look like this:

```
1010101010101010
```

The above equals AAAA in hexadecimal.

The last point referenced after a LINE statement is executed is point (*x2,y2*). If the relative form is used for the second set coordinates, it is relative to the first pair of coordinates. For instance, the following statement could be used to draw a line from point (150,100) to point (140,80).

```
LINE (150,100)-STEP (-10,-20)
```

Full Line Clipping is now performed. In BASIC 1.0, when a line was drawn outside of the Screen it was not properly truncated when it reached the screen's edge. Instead, it was folded back into the screen in a bizarre fashion. With Line Clipping, points plotted outside of the Screen or VIEW port limits do not appear, and lines that intersect the screen or VIEW port limits will appear, cross the screen (or VIEW port) and then disappear correctly at the other end.

Examples: The simplist form of LINE statement looks like this:

```
LINE -(X2, Y2)
```

This would draw a line from the last point referenced to the point (X2,Y2) using the current foreground color.

A starting point can also be included. The following line will draw a diagonal line down the screen from the upper left corner:

```
LINE (0,0)–(319,199)
```

The following statement will draw a line across the screen:

```
LINE (0,90)–(319,90)
```

The next statement shows how the *color* argument is used to indicate the color of the line:

```
LINE (5,5)–(10,10),2
```

The current color #2 will be used to draw the line. Random colors can also be requested:

```
100 SCREEN 1,0,0,0: CLS
200 LINE –(RND*319, RND*190), RND*4
```

The following example shows how an alternating pattern of lines may be achieved:

```
100 SCREEN 1,0,0,0: CLS
200 FOR J=0 TO 319
300 LINE (J,0)–(J,199),J AND 1
400 NEXT
```

The next example shows how the B argument can be used to draw a box in the current foreground color:

```
LINE (0,0)-(150,150),,B
```

Or the color may be specified; the following example shows how a colored box can be drawn:

```
LINE (0,0)-(150,150),2,BF
```

The last example shows how the *style* parameter can be used to draw a dashed line upper left hand corner of the screen to the center:

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

LINE INPUT Statement

Syntax: **LINE INPUT**[;] [*"prompt"*;] *stringvar*

Purpose: Reads a line (up to 255 characters) that is entered from the keyboard and places it in a string variable, without the use of delimiters.

Comments: Variables are defined as follows:

"prompt" is a string constant that will be displayed on the screen before the line is entered. A question mark is only displayed if it is included in the prompt string.

stringvar represents the name of a string variable or array element to which the line will be assigned. All input from the end of the prompt to the Enter considered to be the line, with any trailing blanks ignored.

If a semicolon is placed immediately after LINE INPUT, then pressing the <ENTER> key to end the line does not echo a carriage return/line feed sequence on the screen.

LINE INPUT can be aborted by pressing <Ctrl> <Break>. BASIC will return to command level. To resume execution at the LINE INPUT, enter CONT.

Examples: See the example in the following section, "LINE INPUT# Statement."

LINE INPUT# Statement

Syntax: LINE INPUT *#filenum, stringvar*

Purpose: Reads a line (up to 255 characters), without delimiters, from a sequential file and places it in a string variable.

Comments: *filenum* is the number associated with the file when it was OPENed.

stringvar represents a string variable or array element name that will be assigned to the line.

LINE INPUT # reads all characters in a sequential file from one carriage return to the next. The initial carriage return characters are returned as part of the string.

LINE INPUT # is especially useful if file lines have been broken into fields, or if a BASIC program saved in ASCII characters is being read as data by another program.

LINE INPUT # can also be used for random files.

Examples: The example below uses LINE INPUT to get information likely to contain commas or other delimiters from the keyboard. The information is then written to a sequential file, then read back out of the file with LINE INPUT #.

```
100 OPEN "FILE" FOR OUTPUT AS #2
200 LINE INPUT "Address?";ADD$
300 PRINT #2, ADD$
400 CLOSE 2
500 OPEN "FILE" FOR INPUT AS #2
600 LINE INPUT #2, ADD$
700 PRINT ADD$
800 CLOSE 2
RUN

Address?
```

You might respond with with MENLO PARK, CA 94305. The program continues:

```
Address? MENLO PARK, CA 94305

MENLO PARK, CA 94305
Ok
```

LIST Command

Syntax: **LIST** [*line1*] [−[*line2*]] [,*filespec*]

Purpose: Lists the program in memory on the screen or another specified device.

Comments: *line1, line2* are line numbers ranging from zero to 65529. *line1* is the first line, and *line2* is the final listed line. The current line may be specified by using a period for either *line1* or *line2*.

filespec is a string expression for the output file specification. If *filespec* is omitted, the lines are displayed on the screen.

Pressing <Ctrl> <Break> will interrupt a LISTing, either on the screen or the printer.

If the line range is omitted, the entire program will be listed.

A dash (-) may be included in a line range for one of three purposes:

line1	means list from *line1* to the end of the program.
line2	means list all lines from the beginning of the program through *line2*.
line1-line2	means list all lines from *line1* through *line2*, inclusive.

COMMANDS STATEMENS

Examples:

```
LIST
```

The entire program will be listed on the screen.

```
LIST 100, "SCRN:"
```

Line 100 will be listed on the screen.

```
LIST 100-200, "LPT1:"
```

Lines 100 through 200 will be listed on the printer.

```
LIST 200-, "COM1:1200,N,8"
```

Lists all lines from 200 through the end of the program to the first communications adapter at 1200 bps, no parity, 8 data bits, 1 stop bit.

```
LIST -500, "PROG2"
```

Lists form the first line through line 500 to a disk file named "PROG2"

LLIST Command

Syntax: **LLIST** [*line1*][–[*line2*]]

Purpose: Lists all or part of the program in memory on the printer (LPT1:).

Comments: The line number range for LLIST is the same as that for LIST.

After a LLIST is executed, BASIC returns to command level.

Examples:

```
LLIST
```

Prints the entire program.

```
LLIST 100
```

Prints line 100.

```
LLIST 100-200
```

Prints lines 100 through 200.

```
LLIST 300-
```

Prints all lines from 300 through the end of the program.

```
LLIST -500
```

Prints the first line through line 500 of the program.

LOAD Command

Syntax: **LOAD** *filespec*[,R]

Purpose: Transfers a program from a specified device into memory, and optionally runs it.

Comments: *filespec* is a string expression representing the file specification. LOAD closes any open files and deletes any variables and program lines that may be in memory, then loads the specified program into memory.

If the R option is included, BASIC then immediately executes the program. If R is omitted. BASIC returns to direct mode after loading the program. When R is included, all open data files remain open. Thus, LOAD with the R option can be used to chain a number of programs (or program segments). Information can be passed between the programs via data files.

If the filename is eight characters or less, and if no extension is included with the filename, BASIC adds the extension .BAS to the filename.

Examples:

```
LOAD "PROG"
```

Loads the program named PROG into memory.

```
LOAD "PROG5",R
```

Loads and runs the program named PROG5.

```
LOAD "B:REPOST.DAT"
```

Loads the file REPORT.DAT from Drive B. The extension need only be specified if there is a conflict with, say, a file named REPORT.BAS.

LOC Function

Syntax: *v*=**LOC**(*filenum*)

Purpose: Returns the current file position.

Comments: *filenum* is the number under which the file was opened.

When LOC is applied to a random file, it returns the record number of the last record read from or written to the file.

When the argument is a sequential file, LOC returns the number of records read or written since the file was OPENed. (A record is 128 bytes long.)

When a sequential file is opened for input, BASIC reads the first sector, so LOC will return 1 even before any input is done.

For a communications file, LOC returns the number of pending characters in the input buffer. The default input buffer size is 256 characters, but this can be changed with the /C: option of the BASIC command. If the buffer contains more than 255 characters, LOC returns 255. Since a string may not be longer than 255 characters, it is unnecessary to test a string's size before reading data into it. If less than 255 characters remain in the buffer, then LOC returns the actual number.

Examples: The following stops the program when 10 records have been read.

```
500 IF LOC(1)>10 THEN STOP
```

LOCATE Statement

Syntax: **LOCATE** [*row*][,[*col*] [,[*cursor*][,[*start*] [,*stop*]]]]

Purpose: Positions the cursor on the screen. Optional arguments turn the blinking of the cursor on and off and specify the size of the blinking cursor.

Comments: *row* is a numeric expression from one to 25 that indicates the screen line number where the cursor should be placed.

col is a numeric expression from 1 to 40 (or 80, depending upon screen width) that specifies the screen column number where you wish to place the cursor.

cursor is a value that indicates whether the cursor should be visible or not. Zero indicates invisible; 1 (one) means visible.

start is the cursor's starting scan line, and must be a numeric expression from zero to 31.

stop is the cursor's ending scan line, and also must be a numeric expression from zero to 31.

start and *stop* are used to make the cursor a specified size by indicating the starting and ending scan lines. Scan lines are numbered from zero at the top of each character position. The bottom scan line is 7. If *start* but not *stop* is given, *stop* is assigned the value of *start*.

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

While a program is running, the cursor is normally off. LOCATE,,1 can be used to turn it back on.

Normally, BASIC will not print on line 25. But you can turn off the soft key display with KEY OFF, and then use LOCATE25,1:PRINT...to write on line 25.

Examples:

```
100 LOCATE 1,1
```

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

```
200 LOCATE ,,1
```

Makes a blinking cursor visible; the position is not unchanged.

```
300 LOCATE ,,,7
```

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

```
400 LOCATE 2,1,1,0,7
```

The cursor is moved to line 2, column 1. The cursor becomes visible, covering the entire character cell starting at scan line 0 and ending on scan line 7.

LOF Function

Syntax: *v*=**LOF**(*filenum*)

Purpose: Returns the number of bytes used by a given file (its length).

Comments: *filenum* is the file number assigned to the file when it was OPENed.

In BASIC 1.0, LOF will return a multiple of 128. For example, if the actual data in the file is 257 bytes, the number 384 will be returned. For non-BASIC files and for files created in BASIC 2.0, LOF returns the actual length of the file.

For a communications file, LOF will return the amount of the remaining space in the input buffer. The default size of the input buffer is 256 bytes, however this can be changed with the /C: option of the BASIC command.

Examples: The following example displayes the length of the file named RECORDS on the screen.

```
100 OPEN "RECORDS" AS #1
200 PRINT LOF(1)
```

LOG Function

Syntax: *v*=**LOG**(*x*)

Purpose: Returns the natural logarithm of a given *x*.

Comments: *x* must be a positive numeric expression. (The natural log is the log to the base *e*.)

Examples: The following example calculates the natural log of the expression 100−5:

```
PRINT LOG(100-5)
 4.553877
Ok
```

COMMANDS STATEMENS

LPOS Function

Syntax: *v*=**LPOS**(*n*)

Purpose: Returns the current position of the print head in the output buffer.

Comments: *n* is a numeric expression that indicates which printer is being tested.

0 or 1	LPT1:
2	LPT2:
3	LPT3:

The returned value does not necessarily represent the physical position of the print head *on the printer.*

Examples: If the line length is more than 65 characters a carriage return character is sent to the printer.

```
500 IF LPOS(0)>65 THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

Syntax: **LPRINT** [*list of expressions*] [;]

LPRINT USING *v$; list of expressions* [;]

Purpose: Prints specified output on the printer.

Comments: The *list of expressions* specifies the numeric and/or string expressions to be printed. These expressions must be separated by commas or semicolons.

v$ is a string constant or variable that designates the format to be used for printing. Output formatting is explained in detail under "PRINT USING Statements."

PRINT and PRINT USING output to the screen, and LPRINT and LPRINT USING output to the printer in the same way.

The line length may be changed with a WIDTH "LPT1:" statement.

Printing and processing are asynchronous. If you do a form feed (LPRINT CHR$(12);) followed by another LPRINT and the printer takes more than 10 seconds to execute the form feed, you may get a "Device Timeout" error message on the second LPRINT. Doing the following will avoid this:

```
10 ON ERROR GOTO 2500
•
•
•
2500 IF ERR=24 THEN RESUME
```

Testing ERL will tell you whether the timeout was caused by an LPRINT statement.

Examples: This example will print a heading and a list of numbers. This first number will appear as three digits preceded by a blank space; the "AMOUNT" will be printed as a dollar amount with a decimal point.

```
500 LPRINT "NUMBER AMOUNT"
550 LPRINT USING "###   ###.##";NUM,AMT
```

COMMANDS STATEMENS

LSET and RSET Statements

Syntax: **LSET** *stringvar=x$*

RSET *stringvar=x$*

Purpose: Positions data in a random file buffer (in preparation for a PUT statement).

Comments: *stringvar* represents a variable name defined in a FIELD statement.

x$ is a string expression designating the information to be placed into the *stringvar* field.

If *x$* requires less length than specified for *stringvar* in the FIELD statement, LSET left-justifies *x$* in the *stringvar* field, and RSET right-justifies the string, with spaces being used to pad the extra positions. If *x$* is longer than *stringvar,* characters are truncated from the right.

Numeric values must be converted to strings before being LSET or RSET. Refer to the MKI$, MKS$, and MKD$ Functions.

LSET or RSET may also be used to left- or right-justify a string variable not defined in a FIELD statement in a given field. For example, the following lines right-justify the string Y$ in a 10-character field, which can be useful when formatting printed output.

```
200 X$=SPACE$(10)
250 RSET X$=Y$
```

Examples: This example converts the numeric value NUM into a string, and left-justifies it in the field N$ in preparation for a PUT statement:

```
100 LSET N$=MKS$(NUM)
```

MERGE Command

Syntax: **MERGE** *filespec*

Purpose: Merges the lines of an ASCII disk file into those of the program in memory.

Comments: *filespec* is a string expression that specifies the disk file.

The disk is searched for the specified file. If found, the program lines of the disk file are merged with the lines in memory. If any lines in the disk file program have the same line number as those in memory, the lines from the disk file will replace the corresponding memory lines.

If the program being merged was not saved in ASCII format (with the A option of the SAVE command), a "Bad file mode" message will appear.

Examples: This command inserts, by sequential line number, all lines in the program NUMBERS into the program currently in memory.

```
MERGE "NUMBERS"
```

MID$ Function and Statement

Syntax of the function:

*v$*=**MID$**(*x$*,*n*[,*m*])

of the statement:

**MID$**(*v$*,*n*[,*m*])=*y$*

Purpose: The MID$ function returns the specified portion of a given string. The MID$ statement replaces part of one string with another string.

Comments: In the function specification *v$*=MID$(*x$*,*n*[,*m*]): *x$* is string expression.

n is an integer expression from one to 255.

m is an integer expression from zero to 255.

The function returns a string *m* characters long, starting with the *n*th character of *x$*. If *m* is omitted, or there are less than *m* characters to the right of the *n*th, all characters to the right of the *n*th are returned. If *m* is zero, or if *n* is more than the length of *x$*, then MID$ will return a null string.

In the statement MID$(*v$*,*n*[,*m*])=*y$*:

v$ represents a string variable or array element that will have all or part of its characters replaced.

n is an integer expression from one to 255.

m is an integer expression from zero to 255.

y$ represents a string expression.

Beginning at position *n*, characters *v$* are replaced by *m* characters from *y$*. If *m* is not given, all of *y$* is used.

But regardless of whether *m* is omitted or included, the length of *v$* will not change. For instance, if *v$* is 10 characters long and *y$* is 15 characters long, then after the replacement, *v$* will contain only the first 10 characters of *y$*.

If either n or *m* is out of range, an "Illegal function call" message will be displayed.

Examples: The following example uses the MID$ function to select the middle portion of the string J$.

```
100 I$="GOOD"
200 J$="MORNING AFTERNOON EVENING"
300 PRINT I$;MID$(J$,8,10)
RUN
GOOD AFTERNOON
Ok
```

In the next example MID$ statement replaces characters in the string A$.

```
100 A$="KANSAS CITY, MO"
200 MID$(A$,14)="KS"
300 PRINT A$
RUN
KANSAS CITY, KS
Ok
```

COMMANDS STATEMENS

MKDIR Statement

Syntax: **MKDIR** *path*

Purpose: Creates a directory on a disk.

Comments: *path* is a valid string expression identifying the directory to be created. The string must not exceed 63 characters.

Examples: This example starts from the root directory and creates a sub-directory called INVEST.

```
MKDIR "INVEST"
```

Starting from the root directory, a sub-directory called TAX is created under the directory INVEST.

```
MKDIR "INVEST\TAX"
```

Still starting from the root directory, a sub-directory IRA is created under the directory TAX.

```
MKDIR "INVEST\TAX\IRA"
```

Starting from the root directory and creates a sub-directory called INSURE.

```
MKDIR "INSURE"
```

Make INSURE the current directory, and two sub-directories called HOUSE and CAR are created.

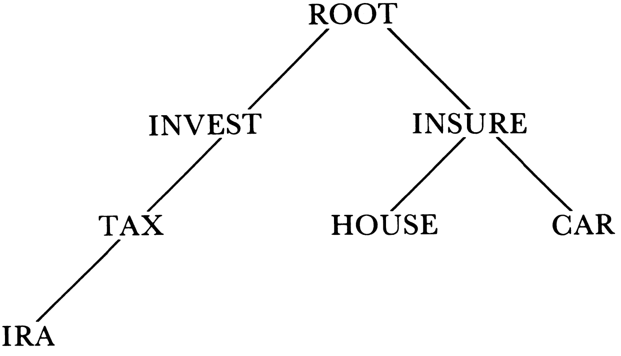
```
CHDIR "INSURE"
MKDIR "HOUSE"
MKDIR "CAR"
```

You can create the same structure in the difference way.

YOU TYPE:

```
MKDIR "INSURE\HOUSE"
MKDIR "INSURE\CAR"
```

From the root directory, the same structure was created.



COMMANDS STATEMENS

MKI$, MKS$, MKD$ Functions

Syntax: *v$*=**MKI$**(*integer expression*)

*v$*=**MKS$**(*single-precision expression*)

*v$*=**MKD$**(*double-precision expression*)

Purpose: Convert numeric values to strings.

Comments: The LSET and RSET statements are used to place string data into a random file buffer. Thus, numeric values must be converted to strings before LSET or RSET can be used to move them.
MKI$ makes an integer value into a 2-byte string, MKS$ is used to convert a single-precision number to a 4-byte string, and MKD$ converts a double-precision number into an 8-byte string.

These functions differ from STR$ in that they do not actually change the data, only the way it is interpreted by BASIC. Refer also to the CVI, CVS, and CVD Functions.

Examples: The following example uses a random file (#1) with fields defined in line 100. The first field, A$, will hold a numeric value, AMT. Line 200 converts AMT to a string with MKS$ and uses LSET to place this string in the random file buffer. Line 300 places a string in the buffer, then line 400 writes the data from the buffer to the file.

```
100 FIELD #1,4 AS A$, 20 AS NAME$
200 LSET A$=MKS$(AMT)
300 LSET NAME$=CUSTNAM$
400 PUT #1
```

NAME Command

Syntax: **NAME** *filespec* AS *filename*

Purpose: Assigns a new name to a disk file.

Comments: *filespec* specifies an existing disk file whose name is to be changed.

filename specifies the new filename.

filespec must designate an existing file and *filename* must not be currently used on the disk.

After a NAME command is executed, the file exists on the same disk, in the place, with the new name.

Examples:

```
NAME "ACCT.BAS" AS "ACCPAY.BAS"
```

In this example, the disk file named ACCT.BAS will now be named ACCPAY.BAS.

NEW Command

Syntax NEW

Purpose: Deletes the program currently in memory and clears any variables.

Comments: NEW is normally used to clear memory prior to entering a new program. NEW closes all files and turns tracing off.

Examples:

```
NEW
Ok
```

Any program in memory is now deleted, and any variables cleared.

OCT$ Function

Syntax: *v$*=**OCT$**(*n*)

Purpose: Returns a string representing the octal value of a given decimal argument.

Comments: *n* is a numeric expression from −32768 to 65535.

If *n* is negative, OCT$ converts the argument to the two's complement form, which means that OCT$(−*n*) is the same as OCT$(65536−*n*).

Examples: The following example shows how the OCT$ function can be used.

```
500 OCTOUT$=OCT$(24)
600 PRINT OCTOUT$; " OCTAL IS DECIMAL 24"
RUN
30 OCTAL IS DECIMAL 24
Ok
```

COMMANDS STATEMENS

ON COM(n) Statement

(BASICA only)

Syntax: **ON COM**(*n*) GOSUB *line*

Purpose: Specifies the line number of a trapping routine for BASIC to branch to when there is information coming into a designated communications buffer.

Comments: *n* represents the number of the communications adapter (1 or 2).

line is the line number of the start of the trap routine. If *line* is zero, trapping of communications activity for the specified adapter is disabled.

The COM(*n*) ON statement must be executed to activate trapping by the ON COM(*n*) statement. After a COM(*n*) ON statement, if *line* is non-zero in the ON COM(*n*) statement, then on each new statement, BASIC checks to see if any characters have come in to the specified communications buffer. If so, BASIC branches to the specified *line*.

If a COM OFF statement has been executed for the communications channel, trapping is not performed and is not remembered.

If a COM(*n*) STOP is executed, no trapping will take place for the communications channel, but it will be performed as soon as a COM(*n*) ON statement is executed.

When an event trap occurs, automatic COM(*n*) STOP is executed so that recursive traps cannot take place.

The RETURN from the trap routine automatically executes a COM(*n*) ON statement unless an explicit COM(*n*) OFF was done with in the trap routine.

Trapping does not happen unless BASIC is executing a program. When an error trap (resulting from an ON ERROR statement) takes place, all trapping is automatically disabled, including ERROR, STRIG(*n*), COM(*n*), and KEY (*n*).

Examples: This example specifies a branch to a trap routine for the internal serial interface at line 900.

```
110 ON COM(1) GOSUB 900
120 COM(1) ON
•
•
•
900 REM characters coming in
•
•
•
990 RETURN 300
```

COMMANDS STATEMENS

ON ERROR Statement

Syntax: **ON ERROR GOTO** *line*

Purpose: Activates error trapping and specifies the first line of the error handling subroutine.

Comments: *line* represents the line number of the first line of the error handling routine.

Once error trapping has been activated, all errors detected (including those in direct mode) will cause immediate execution of the specified error handling subroutine.

To deactivate error trapping, execute an ON ERROR GOTO 0 statement. Subsequent errors will cause an error message to be printed and execution halted. If this statement appears in an error trapping subroutine, BASIC prints the message for the error that activated the trap. All error trapping subroutines should execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Error trapping does not occur within an error handling subroutine. If an error occurs within an error handling subroutine, the BASIC error message is printed and execution stops.

The RESUME statement is used to exit the error trapping routine.

Examples:

```
100   ON ERROR GOTO 1000
•
•
•
•
1000 PRINT "ERROR IN LINE";ERL
1010  RESUME NEXT
```

Line 1000 and 1010 are the error handling subroutine that displays the line number where any error occurs.

ON...GOSUB and ON...GOTO Statements

Syntax: **ON** *n* **GOTO** *line*[,*line*]...
ON *n* **GOSUB** *line*[,*line*]...

Purpose: Branches to one of several specified line numbers, depending on the value of a given expression.

Comments: *n* represents a numeric expression that is rounded to an integer, if necessary, and can range from zero to 255.

line is the line number to which to branch.

The value of *n* determines which *line* will be used for branching. For instance, if the value of *n* is 3, the third line number will be the destination of the branch.

In an ON...GOSUB statement, each line number in the list must be the first line of a subroutine, which ends with a RETURN statement directing the program back to the line after the ON...GOSUB statement.

If *n* is zero or more than the number of *lines* in the list, BASIC will continue with the next executable statement.

Examples: In the first example, the program will branch to line 200 if J-1 equals 1, to line 300 if J-1 equals 2, to line 400 if J-1 equals 3, and to line 500 if J-1 equals 4. If J-1 is zero or more than 4, the program goes on to the next statement.

```
100 ON J-1 GOTO 200,300,400,500
```

The next example shows how an ON GOSUB statement can be used.

```
10 REM display the menu
20 PRINT "Routine 1"
25 PRINT "Routine 2"
30 PRINT "Routine 3"
35 PRINT "Routine 4"
40 INPUT "Which routine do you want?", CHOICE
50 ON CHOICE GOSUB 100, 200, 300, 400
60 GOTO 10
70 REM redisplay menu after return routine
100 REM start of routine 1
•
•
•
190 RETURN
200 REM start of routine 2
•
•
•
```

ON KEY(n) Statement

(BASICA only)

Syntax: **ON KEY**(*n*) GOSUB *line*

Purpose: Specifies a line number for BASIC to branch to when a given function or cursor control key is pressed.

Comments: *n* is a numeric expression ranging from 1 to 20 that specifies the key to be trapped, as follows:

1-10 function keys F1 to F10
11 Cursor Up
12 Cursor Left
13 Cursor Right
14 Cursor Down
15-20 keys defined by the syntax:
KEY *n*, CHR$(*s*)+CHR$(*t*)
(See "KEY(*n*)" and "KEY" statements in this chapter. For BASIC 2.0 and later only.)

line is the line number of the beginning of the trap routine for the indicated key. Trapping of the key stops when *line* is zero.

The KEY(*n*) ON statement must be executed to activate trapping by the ON KEY(*n*) statement. After KEY(*n*) ON, if *line* is non-zero in the ON KEY(*n*) statement, then on each new statement, BASIC checks to see whether the specified key was pressed. If so, BASIC branches to the specified *line*.

A KEY(*n*) OFF statement will stop trapping for he specified key. If the key is pressed, the event is not noted.

If KEY(*n*) STOP is executed, no trapping takes place for the designated key. However, if the key is pressed, the event is remembered, and an immediate trap will take place if KEY(*n*) ON is executed.

When an event trap occurs, KEY(*n*) STOP is executed so that recursive traps cannot take place. The RETURN from the trap routine automatically executes a KEY(*n*) ON statement unless an explicit KEY(*n*) OFF was done within the trap routine.

Trapping does not happen unless BASIC is executing a program. When an error trap (resulting from an ON ERROR statement) takes place, all trapping is automatically disabled, including ERROR, STRIG(n) COM(n), KEY(*n*), PLAY and TIMER.

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 REUTRN *line* 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".

Examples: The following example shows a trap routine for function key 5.

```
10 ON KEY(5) GOSUB 500
20 KEY(5) ON
•
•
•
500 REM start of trap routine for function key 5
•
•
•
590 RETURN 100
```

This example trap Crtl-Shift L:

```
10 KEY 18, CHR$(&H4+&H1)+CHR$(38)
20 ON KEY(18) GOSUB 1000
30 KEY(18) ON
•
•
•
1000 REM start of trap routine for function key 5
•
•
•
1090 RETURN
```

ON PLAY(n) Statement

(BASICA only)

Syntax: **ON PLAY**(*n*) GOSUB *line*

Purpose: Specifies a line number for BASIC to branch to allow continuous background music during program execution.

Comments: *n* is a numeric expression ranging from 1 to 255 (representing the number of notes to be trapped). If a value outside these limits is entered, an "Illegal function call" error results.

line is the beginning line number of the PLAY trap routine. PLAY trapping stops when *line* is zero.

The PLAY ON statement must be executed to activate trapping by the ON PLAY(*n*) statement. After PLAY ON, if *line* is non-zero in the ON PLAY(*n*) statement, then on each new statement, BASIC keeps track of the music buffer. When the number of the note in the buffer is less than *n*, BASIC branches to the specified line.

A PLAY OFF statement will stop trapping. If a PLAY activity takes place, the event is not noted.

If PLAY STOP is executed, no trapping takes place. However, if a Play activity takes place, the event is remembered, and an immediate trap takes place if PLAY ON is executed.

When an event trap occurs, PLAY STOP is executed so that recursive traps cannot take place. The RETURN from the trap routine automatically executes a PLAY ON statement unless an explicit PLAY OFF was done within the trap routine.

Trapping does not happen unless BASIC is executing a program. When an error trap (resulting from an ON ERROR statement) occurs, all trapping is disabled, including ERROR, STRIG(n), COM(n), KEY(n), PLAY(*n*), and TIMER(n).

The RETURN *line* form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. Use this type of return carefully because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, resulting in such errors as "FOR without NEXT".

> **NOTES:** A PLAY event trap can only take place when PLAY is in the Music Background mode (PLAY "MB..."). PLAY event traps cannot take place when PLAY is running in the Music Foreground mode (PLAY "MF...").
>
> If the Music Background buffer is empty when a PLAY ON statement is executed, no event trapping takes place.
>
> Choose lower values for *n* (for example 3). To high a value (for example 47) causes so many event traps that little time is left to run the remainder of the program.

Example: The following example sets up a trap routine which starts when ten notes are left in background music buffer.

```
100 ON PLAY(10) GOSUB 800
120 PLAY ON
•
•
•
800 REM continuous background music subroutine
•
•
•
950 RETURN 130
```

ON STRIG(n) Statement

(BASICA only)

Syntax: **ON STRIG(*n*)** GOSUB *line*

Purpose: Specifies a line number for BASIC to branch to when a joystick button is pressed.

Comments: *n* can be zero, 2, 4, or 6, and specifies the button to be trapped as follows:

n	*button*
0	A1
2	B1
4	A2
6	B2

line represents the line number of the trap routine. When *line* is zero, trapping is disabled.

The STRIG(*n*) ON statement must be executed to activate trapping by the ON STRIG(*n*) statement. After STRIG(*n*) ON is executed and if *line* is non-zero in the ON STRIG(*n*) statement, then on each new statement BASIC checks to see whether the specified button has been pressed. If so, BASIC branches to the specified *line*.

STRIG(*n*) OFF stops trapping for button *n*. if the buttion is pressed, the event is not noted.

If STRIG(*n*) STOP is executed, no trapping takes place for button *n*, but pressing of the button is remembered, and an immediate trap will take place if STRIG(*n*) ON is executed.

When an event trap occurs, STRIG(*n*) STOP is executed so that recursive traps cannot take place. The RETURN from the trap routine automatically executes a STRIG(*n*) ON statement unless an explicit STRIG(*n*) OFF was done within the trap routine.

Trapping does not happen unless BASIC is executing a program. When an error trap (resulting from an ON ERROR statement) takes place, all trapping is automatically disabled, including ERROR, STRIG(*n*), COM(*n*) and KEY(*n*).

STRIG(*n*) ON will activate the interrupt routine that checks the status for the specified joystick button. Downstrokes that activate trapping will not set function STRIG(0), STRIG(2), STRIG (4), or STRIG(6).

RETURN *line* can be used to go back to the BASIC program at a specific line number. Use of this non-local return must be done carefully, however, since WHILEs, FORs, or any other GOSUBs active at the time of the trap will remain active.

Examples: The following example shows a trapping routine for the button on the first joystick.

```
10 ON STRIG(0) GOSUB 300
20 STRIG(0) ON
•
•
•
300 REM 1st button subroutine
•
•
•
310 RETURN
```

COMMANDS STATEMENS

ON TIMER(n) Statement

Syntax: **ON TIMER(*n*)** GOSUB *line*

Purpose: Specifies a line number for BASIC to branch to after a specified period of time has elapsed.

Comments: *n* is a numeric expression ranging from 1 to 86,400 (representing 1 second through 24 hours). If a value outside these limits is entered, an "Illegal function call" error results.

line is the beginning line number of the TIMER trap routine. TIMER trapping stops when *line* is zero.

The TIMER ON statement must be executed to activate trapping by the ON TIMER(*n*) statement. After TIMER ON, if *line* is non-zero in the ON TIMER(*n*) statement, then on each new statement, BASIC keeps track of the passing seconds. When *n* seconds are counted, BASIC branches to the specified line. After the event trap BASIC again begins counting from 0.

A TIMER OFF statement will stop trapping. If a TIMER activity takes place, the event is not noted.

If TIMER STOP is executed, no trapping takes place. However, if a TIMER activity takes place, the event is remembered, and an immediate trap takes place if TIMER ON is executed.

When an event trap occurs, TIMER STOP is executed so that recursive traps cannot take place. The RETURN from the trap routine automatically executes a TIMER ON statement unless an explicit TIMER OFF was done within the trap routine.

Trapping does not happen unless BASIC is executing a program. When an error trap (resulting from an ON ERROR statement) occurs, all trapping is automatically disabled, including ERROR, STRIG(n), COM(n), KEY(n), PLAY(n) and TIMER(*n*).

The RETURN *line* form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. Use this type of return carefully because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, resulting in such errors as "FOR without NEXT".

Example: The following example displays the time of day every 1 minute:

```
100 ON TIMER(60) GOSUB 500
120 TIMER ON
•
•
•
500 REM The TIMER trap routine from line 100
510 CRNTR=CSRLIN 'save current row
520 CRNTC=POS(0) 'save current column
530 LOCATE 1,1:PRINT TIME$
540 LOCATE CRNTR,CRNTC 'restore row and column
590 RETURN
```

OPEN Statement

Syntax:

OPEN *filespec* [FOR *mode*] AS [#]filenum [LEN=*recl*]
OPEN *path* [FOR *mode*] AS [#]*filenum* [LEN=*recl*]

Alternative form:

OPEN *mode2*, [#]*filenum*, *filespec* [,*recl*]

OPEN *mode2*, [#] *filenum*, *path* [,*recl*]

Purpose: Permits I/O operations to a file or device.

Comments: *mode* is one of the following:

OUTPUT means sequential output mode. In this context, "output" signifies writing data *to the file*.

INPUT means sequential input mode. "Input" means using the file data *as input*, i.e., "heading" the file.

APPEND means sequential output mode beginning at the end of an existing file.

If *mode* is omitted, random access is the default.

mode2 in the alternate form, is a string expression whose first character is one of the following:

- O means sequential output mode
- A means sequential output mode beginning at the end of an existing file.
- I means sequential input mode
- R means random I/O mode

For either format:

filenum is an integer expression whose value may range from one to 255.

filenum is the number associated with the file as long as it is open and is used by other I/O statements to refer to it.

filespec is a string expression that specifies the file.

path is a string expression up to 63 characters long. See "Introduction" and "Naming Files" in Chapter 5.

recl is an optional integer expression that is used to set the record length, and may range from one to 32767. In BASIC 1.0, you cannot use *recl* for sequential files. But in BASIC 2.0 and later, *recl* is valid for sequential files. The default record length is 128 bytes. *recl* may not be more than the value set by the /S: switch of the BASIC command, if the /I switch is not specified.

OPEN allocates an I/O buffer to the file or device and specifies the mode of access for which the buffer will be used.

An OPEN statement must be executed before any I/O operation may be done to a device or file with any of the following statements, or any other statement or function that requires a file number:

PRINT #	WRITE #
PRINT # USING	INPUT$
INPUT #	GET
LINE INPUT #	PUT

GET and PUT are used with random files (or communications files). A disk file can be either random or sequential, and a printer can be opened for either random or sequential output. But, all other devices may be opened for sequential operations only.

BASIC will normally add a line feed after each carriage return (CHR$(13)) that is sent to a printer, but, if you open a printer as a random file with width 255, this line feed is suppressed.

APPEND is valid only for disk files. The file pointer is initially set to the end of the file and the record number is set to the last record. PRINT # or WRITE # will then add records to the file, extending it.

At any given time, it is possible to have a particular file open under more than one file number, allowing different modes to be used for different purposes. Or, for program clarity, you might use different file numbers for different modes of access. Each file number is associated with a different buffer, so take care if you are writing using one file number and reading using a different one.

However, note that a file that is already open may not be opened for sequential output or append.

If a nonexistent file is OPENed for input, a "File not found" message appears. If a file that does not exist is OPENed for output, append, or random access, a file with the specified name is created.

Examples: Either of the following statements opens a file named EXFILE for sequential output (that is, data will be written to the file).

```
100 OPEN "EXFILE" FOR OUTPUT AS #1
```

or

```
100 OPEN "0", #1, "EXFILE"
```

In the above examples, note that opening the file for output destroys any data that may be in the file, since this format causes data to be written to the file starting at the beginning. If you wish to *add to existing data,* open the file for append, as follows:

```
100 OPEN "EXFILE" FOR APPEND AS #1
```

or

```
100 OPEN "A", #1, "EXFILE"
```

Either of the following two statements will open a disk file named EXFILE for random input and output, with a record length of 256.

```
100 OPEN "EXFILE" AS 1 LEN=256
```

or

```
100 OPEN "R",1,"EXFILE",256
```

The following statements show how an OPEN statement may alternately be written using a string variable for the filename:

```
100 FILE$="STRFILE"
200 OPEN FILE$ FOR APPEND AS 2
```

This example shows underlining.

```
10 OPEN "LPT1:" AS #1
20 WIDTH #1,255
30 PRINT #1, "Underline this line"
40 WIDTH #1, 80
50 PRINT #1, STRING$ (19, "_")
```

As described in Chapter 5, BASIC 2.0 and later allows you to follow paths to files. The OPEN command can be used with paths. For example:

```
100 OPEN "INVEST\TAX\IRA" FOR OUTPUT AS #1
```

or

```
100 OPEN "O",#1,"INVEST\TAX\IRA"
```

Either of the above statements opens the file named "IRA" for sequential output on the default device in the directory TAX. Either of the following statements will open a file named TFILE in the STEP1 directory on driver A for random input and output. The record length is 256.

```
200 OPEN "A:STEP1\TFILE" AS 1 LEN=256
```

or

```
200 OPEN"R",1,"A:STEP1\TFILE",256
```

OPEN"COM... Statement

Syntax: **OPEN"COM***n*:[*speed*] [,*parity*] [,*data*] [,*stop*] [,RS] [,CS[*n*] [,DS]*n*] [,CD]*n*]] [,LF] [,PE]" AS [#]*filenum* [*LEN=number*]

Purpose: Opens a communications file.

Comments: *n* may be either 1,or 2, and indicates the number of serial interface.

speed is an integer constant that specifies the transmit/receive rate in bits per second (bps). Valid *speeds* are 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600, with the default 300 bps.

parity is a one-character constant that specifies the transmit/receive parity as follows:

S =SPACE: Parity bit always transmitted/received as a space bit value zero.

O =ODD: Odd transmit parity, add receipt parity check.

M =MARK: Parity bit always transmitted/received as a mark bit value one.

E =EVEN: Even transmit parity, even receipt parity check.

N =NONE: No transmit parity, and no receipt parity check.

The default is EVEN (E) parity and receive checking.

data is an integer constant specifying the number of transmit/receive data bits. Valid values are 4, 5, 6, 7, or 8, with the default 7 bits.

stop is an integer constant that specifies the number of stop bits. Valid values are 1 or 2. The default is two stop bits for 75 and 110 bps, and one stop bit for all other *speeds*. If you specify 4 or 5 for *data*, a *stop* of 2 will mean 1-1/2 stop bits.

filenum is an integer expression whose result to a valid file number, which is then associated with the file as long as it is open and is used by later communications I/O statements to refer to the file.

number is the maximum number of bytes that can be read from the communication buffer with a GET or PUT statement. The default is 128 bytes.

OPEN"COM... allocates an I/O buffer in the same way as OPEN for disk files, supporting RS232 asynchronous communication with other computers and peripherals.

Only one file number at a time may be assigned to a communications device.

The RS, CS, DS, CD, LF and PE options affect line signals in the following ways:

RS will suppress RTS (Request To Send).

CS[n] is used to control CTS (Clear To Send).

DS[n] is used to indicated DSR (Data Set Ready).

CD[n] is used to control CD (Carrier Detect), which is also called the RLSD (Received Line Signal Detect).

LF sends a line feed after carriage return.

PE allows parity checking.

NOTE: *speed, parity, data,* and *stop* are positional parameters, that is, RS, CS, DS, CD, LF and PE are not positional.

RTS (Request To Send) line is turned on when an OPEN"COM... statement is executed unless the RS option is included.

n in the CS, DS, and CD options specifies the number of milliseconds to wait for the signal before returning a "Device Timeout" error. *n* can be any number from zero to 65535. If *n* is omitted or zero, the line status is not checked.

The defaults are n=1000 for CS and DS, and n=zero for CD. If RS is specified, the default for CS is zero.

What this means is that usually I/O statements to a communications file will fail if the CTS or DSR signals are off. The system will wait one second before returning a "Device Timeout" message. The CS and DS options let you ignore these lines, or specify the waiting time before the timeout.

Usually Carrier Detect (CD or RLSD) is ignored when OPEN"COM... is executed. The CD option lets you test this line by including the *n* argument, in the same way as CS and DS. If *n* is omitted or zero, then Carrier Detect is not checked (which is like omitting the CD option).

The LF parameter is intended to be used communication files as a means of printing to a serial line printer. When LF is specified, a line feed character (hex 0A) is automatically sent after each carriage return character (hex 0D). Note that INPUT # and LINE INPUT #, when reading from a communications file opened with the LF option, stop on encountering a carriage return, and ignore any line feeds.

The PE option allows parity checks. Using this parameter will yield a "Device I/O error" message for each parity error. It will also turn on the high order bit for 7 or fewer data bits. The default is no parity checks. Note that the PE option has no effect on framing and overrun errors. These will always turn the high order bit on and result in a "Device I/O error".

Any coding errors within the string expression starting with *speed* cause a "Bad file name" error. No indication is given as to which parameter is in error.

Specifying 8 data bits means you must specify parity N. If you specify 4 data bits, then you must specify a parity, i.e., N parity is invalid. BASIC uses all 8 bits of a byte to store numbers, so if you are transmitting or receiving numeric data, 8 data bits must be specified. (Note that this is not so if numeric data is being sent as text.)

Examples: The following statement opens COM1: for communications as file #1 with all defaults, meaning, the speed of 300 bps, even parity, and 7 data bits with one stop bit.

```
100 OPEN "COM1:" AS 1
```

The following statement opens COM1: for communication at 1200 bps. Parity and numbers of data bits and stop bits will be the defaults.

```
100 OPEN "COM1:1200" AS #1
```

COM1: is opened as File 1 for asynchronous I/O at 600 bps, with no parity produced or checked, 8-bit bytes sent and received, and 1 stop bit transmitted.

```
100 OPEN "COM1:600,N,8" AS #1
```

The next opens COM1: at 1800 bps with no parity, eight data bits, and CS, DS, and CD being checked.

```
10 OPEN "COM1:1800,N,8,,CS,DS,CD" AS #1
```

The following statement opens COM1: at 600 bps with the defaults of even parity and seven data bits and with parity checking. RTS is sent, and Device Timeout is given if DSR is not seen within five seconds.

```
50 OPEN "COM1:600,,,,CS,DS5000,PE" AS #1
```

Note that the commas are used to indicate the position of the positional arguments *parity*, *start*, and *stop*, even though values for them are not specified.

An OPEN statement can be used with an ON ERROR statement to make certain that a modem is working properly before sending any data. For instance, the following program makes sure of a Carrier Detect (CD or RLSD) from the modem before beginning. Line 200 is set to timeout after waiting 10 second. TIMES is set to 3 so we give up if Carrier Detect is not seen within half a minute. Once communication is established, the file is reopened with a shorter wait until timeout.

```
50 TIMES=6
100 ON ERROR GOTO 900
200 OPEN "COM1:1200,N,8,2,CS,DS,CD10000" AS #2
300 ON ERROR GOTO 0
400 CLOSE #2 ' to continue
500 GOTO 2000
•
900 IF ERR=24 THEN GOTO 920
910 ON ERROR GOTO 0
920 TIMES=TIMES-1
930 IF TIMES=0 THEN ON ERROR GOTO 0 ' forget it
940 RESUME
•
•
•
2000 OPEN "COM1:1200,N,8,2,CS,DS,CD5000" AS #1
```

The last example shows a typical way of using communication file to control a serial line printer. The LF argument in the OPEN statement ensures that lines are not printed on top of each other.

```
100 WIDTH "COM1:", 132
200 OPEN "COM1:300,N,8,,CS10000,DS10000,CD10000,
LF" AS #1
```

COMMANDS
STATEMENS

OPTION BASE Statement

Syntax: **OPTION BASE** *n*

Purpose: Specifies the minimum value for array subscripts.

Comments: *n* is either 1 or zero.

The default base is zero.

The OPTION BASE statement must appear *before* any arrays are defined or used. An error will occur if you change the base value once arrays exist.

Examples: If the statement:

```
OPTION BASE 1
```

is executed, the lowest value any array subscript will have is one.

COMMANDS STATEMENS

OUT Statement

Syntax: **OUT** *n,m*

Purpose: Sends a byte to an output port.

Comments: *n* is a numeric expression for the port number, ranging from zero to 65535.

m is a numeric expression for the data to be transmitted, and range from zero to 255.

OUT is the complementary statement to the INP function.

One way OUT can be used is to affect the video output. On some displays, in graphics mode, you may find that the first couple of characters on the line don't show up on the screen. If your display does not have a horizontal adjustment control, the following two statements can be used to shift the display:

```
OUT 980,2:  OUT 981,43
```

This shifts the display two characters to the right with a 40-column width (or 16 points in medium resolution and 32 points in high resolution).

The following statements shift the display to the right five characters with an 80-column width.

```
OUT 980,2:  OUT 981,85
```

Such shifts remain in effect until a WIDTH or SCREEN statement is executed. The DOS MODE command can also be used to shift the display in this way; it has the virtue of remaining in effect until a System Reset is done.

Examples:

```
300 OUT 30,50
```

This sends the value 50 to output port 30.

PAINT Statement

(BASICA only)

Syntax: **PAINT** (*x,y*) [[,*paint*] [,*boundary*][,*background*]]

Purpose: Fills in a screen area with a specified color. Graphics mode only.

Comments: (*x,y*) are the coordinates of a point within the area to be PAINTed, and may be in absolute or relative form. This point is used as a starting point.

paint may be a numeric or string expression. If numeric, it is the color to be PAINTed with, from zero to 3. In medium resolution, this is a color from the current palette as defined by the COLOR statement. Zero means the background color; the default is the foreground color, number 3. In high resolution, *paint* equal to 0 (zero) specifies black, with the default of one meaning white.

When *paint* is a string formula, paint "tiling" is done, as explained below. Tiling is supported only by BASIC 2.0 and later.

boundary specifies the color of the edges of the figure to be PAINTed, and ranges from zero to 3 as described above.

background is a string expression, one byte in length that is used in paint tiling. This parameter is only used in BASIC 2.0 and later.

The figure to be PAINTed is the figure with edges of the *boundary* color. This means that in medium resolution, the figure is *drawn* in the *boundary* color, and is *filled in* with the *paint* color, so we can fill in a border of color 2 with color 1. Visually this might mean a cyan ball with a magenta border.

Since there are only two colors in high resolution, *paint* should be the same as *boundary*. Since *boundary* defaults to equal *paint,* the third parameter is unnecessary in high resolution mode. This means "blacking out" an area until black is specified, or "whiting out" an area until white is indicated.

The PAINT starting point must be inside the figure to be colored. If points outside the screen limits are plotted, they are not drawn and no error occurs. If the specified point is already the *boundary* color, the PAINT statement will have no effect. If *paint* is omitted, the foreground color is used, which is 3 in medium resolution, and 1 in high. PAINT can paint any type of figure; however, jagged edges on a figure will increase the amount of stack space needed to execute the statement. So if much complex coloring will be done, you might want to use CLEAR at the beginning of the program to increase the available stack space.

The PAINT statement allows colored figures to be displayed with very few statements.

BASIC 2.0 and later provide for paint tiling, a technique explained in the following paragraphs.

Tiling is performed when the paint attribute is a string expression in this syntax:

```
CHR$(&Hnn)+CHR$(&Hnn)+CHR$(&Hnn)
```

The tile mask will always be 8 bits wide. It may be up to 64 bytes long. Each byte in the string masks 8 bits along the x axis as points are put down.

Each byte for the string is rotated as required to align along the y axis such that tile byte mask=y mod tile length. This is done so that the tile pattern in a uniform way across the entire screen. The structure of the tile string appears like this:

```
                x increases →
                bit of tile byte
x,y  8 7 6 5 4 3 2 1
0,0  x x x x x x x x Tile byte 0
0,1  x x x x x x x x Tile byte 1
0,2  x x x x x x x x Tile byte 2
           •
           •
           •
0,63 x x x x x x x x Tile byte 63
                     (maximum allowed)
```

In high resolution, each byte of the tile string plots eight points across the screen, or one bit per pixel. A point is plotted at each position in the bit mask that has a value equal to 1. In high resolution, the following statement paint's the screen with x's:

```
PAINT (320,100),CHR$(&H81)+CHR$(&H42)+
      CHR$(&H24)+CHR$(&H18)+ CHR$(&H18)+
      CHR$(&H24)+CHR$(&H42)+CHR$(&H81)
```

COMMANDS STATEMENS

This statement creates a pattern that appears on the screen as:

```
x increases →
1 0 0 0 0 0 0 1 CHR$(&H81) Tile byte 0
0 1 0 0 0 0 1 0 CHR$(&H42) Tile byte 1
0 0 1 0 0 1 0 0 CHR$(&H24) Tile byte 2
0 0 0 1 1 0 0 0 CHR$(&H18) Tile byte 3
0 0 0 1 1 0 0 0 CHR$(&H18) Tile byte 4
0 0 1 0 0 1 0 0 CHR$(&H24) Tile byte 5
0 1 0 0 0 0 1 0 CHR$(&H42) Tile byte 6
1 0 0 0 0 0 0 1 CHR$(&H81) Tile byte 7
```

The length of this mask is eight, with an index of zero through seven. PAINT begins by plotting byte four at coordinates (320,100). The calculation is based on substituting 100 for *y* and eight for *tile length* in the *y* mod tile length formula.

In medium resolution, each byte in the tile pattern describes just 4 pixels, because there are 2 bits per pixel in medium resolution. With medium resolution, every two bits in the tile byte describe one of four possible colors associated with each of the four pixels to be put down.

The binary and hexadecimal values for each color are given in the following chart:

Color palette 0	green	red	brown
Color palette 1	cyan	magenta	white
Color number in binary	01	10	11
Pattern to draw solid line in binary	01010101	10101010	11111111
Pattern to draw solid line in hexadecimal	&H55	&HAA	&HFF

You may sometimes wish to tile paint over an area already painted the same color as two consecutive lines in the tile pattern. Normally, **PAINT** terminates when it encounters two consecutive lines that are the same color as the point being plotted (the point is surrounded by the same color). Use the *background* parameter to keep **PAINT** from quitting. Note, however, that you may not designate more than two consecutive bytes in the tile string that match the background attribute. Specifying more than two results in an **Illegal function call** error message.

Example: The following program demonstrates paint tiling:

```
100 CLS:SCREEN 1,0
200 TIL$=CHR$(&HAA)+CHR$(&HAA)+CHR$(&H55)+
CHR$(&H55)+CHR$(&HFF)+CHR$(&HFF)
300 COLOR 9,0   'palette 0; background light blue
400 VIEW (1,1)-(50,50),0,2
500 GOSUB 1000
600 COLOR ,1   'palette 1
700 GOTO 1300
 •
 •
1000 REM   'jump from line 500
1100 PAINT (40,45),TIL$,1
1200 RETURN
1300 END
```

PEEK Function

Syntax: v = **PEEK**(n)

Purpose: Returns a byte read from the specified memory position.

Comments: n is an integer from zero to 65535. It represents the offset from the current segment as defined by the DEF SEG statement, and specifies the address of the memory location to be read.

The returned value will be an integer from 0 to 255.

PEEK is the complementary function to the POKE statement.

Examples: The example below can be used to know which color is used currently as the foreground color of the characters.

```
100 DEF SEG
110 PRINT PEEK(&H4E)
```

PLAY Statement

(BASICA only)

Syntax: **PLAY** *string*

Purpose: Plays music specified by a character string.

Comments: PLAY implements a concept similar to DRAW, with a "tune definition language" imbedded into a character string.

string is a string expression that consists of single-character music commands.

The music commands are:

A to G — with optional #, +, or − plays the specified note in the current octave. # or + with the note indicates the sharp, and − means the flat. The sharp or flat specification is only allowed if it corresponds to a black piano key. For instance, B# (B sharp) is invalid.

O n — Specifies the octave for the notes that follow. There are 7 octaves, which are numbered zero to 6. Each octave extends from C to B. Octave 3 begins with middle C. The default octave is 4.

N n — Play note n, which may range from zero to 84, since the 7 possible octaves contain 84 notes. n=zero specifies a rest. This is an alternative method of selecting notes to that of specifying the octave (0n) and the note name (A–G), as above.

L n	Sets the length of notes that follow, which is represented by 1/n. n may range from 1 through 64. The possible values of n and the meaning of each are as follows:

L1 (1/1=1)	whole note
L2 (1/2)	half note
L3 (1/3)	one of a triplet of three half notes (1/3 of a 4-beat measure)
L4 (1/4)	quarter note
L5 (1/5)	one of a quintuplet (1/5 of a measure)
L6 (1/6)	one of a quarter note triplet
•	
•	
•	
L64 (1/64)	sixty-fourth note

If you want to change the length only for a single note, the length may follow that note. For example, A16 is the same as L16A.

P n	Rest. n can range from 1 to 64, and specifies the length of a pause in the same way as L (length) indicates the length of a note.

A period following a note, causes the note to be played as a dotted note, with its length is multiplied by 3/2. More than one dot may be used with the length adjusted accordingly. For example, "A.." will play 9/4 as long as specified by L, "A..." will play 27/8 as long, etc. Dots may also be placed after a pause (P) to scale the length of the rest in the same way.

T n	Tempo, indicating the number of quarter notes per minute. n may be from 32 to 255, with the default 120. Under "SOUND Statement" is a table that lists common tempos and the equivalent beats per minute.
MF	Run music in foreground, indicating that each subsequent note will not start until the previous note is finished. Ctrl-Break can be used to exit PLAY. The default state is MF.
MB	Run music in background, indicating that each note is placed in a buffer, which allows the BASIC program to continue while music plays in the background. The buffer will hold up to 32 notes (or rests) at one time.
MN	Music normal, where each note plays 7/8 of the time specified by L (length). This is the default setting for MN, ML and MS and for the commands that are described below.
ML	Music legato, where each note plays the full time set by L (length).
MS	Music staccato, where each note plays 3/4 of the time set by L.
>*n*	Increase one octave and play note *n*. Each time the note is played, it is raised one octave until it reaches the highest octave (6). For example, PLAY ">C" causes note C to be played one octave higher. Thereafter, each time PLAY ">C" is run, it raises C one octave until octave 6 is reached. After reaching the highest octave, note C will continue to play at that level. This command is used only by BASIC 2.0 and later.

<*n* Decrease one octave and play note *n*. Each time the note is played, it is lowered one octave, until the lowest octave is reached (0). For example, PLAY "<C" causes note C to be played one octave lower. Thereafter, each time PLAY "<C" is run, it lowers C one octave, until octave 0 is reached. After reaching the lowest level, note C will continue to play at that level.
This command is used only by BASIC 2.0 and later.

X variable;
Executes the specified string.
In all the music commands the *n* argument may be a constant like 5 or it may be =*variable;* where *variable* is a variable name. A semicolon is required when using a variable this way, and when using the X command. Otherwise, a semicolon is optional between commands, except that is *not* allowed after MF, MB, MN, ML, or MS. Also, blanks in *string* are ignored.

Variables can also be specified in the form of VARPTR$(*variable*), instead of =*variable;*. The VARPTR$ form is the only one that may be used in compiled programs. For instance:

One Method	Alternative
PLAY "XB$;"	PLAY "X"+VARPTR$(B$)
PLAY "L=I;"	PLAY "L="+VARPTR$(I)

You can use X to store a "subtune" in one string and call it recursively with varying tempos or octaves specified by another string.

Examples: The following example plays a chromatic scale.

```
10 A$="CC+DD+EFF+GG+AA+B"
20 FOR I=2 TO 4
30 PLAY "MB MS T150 O=I;L8 XA$;P8"
40 NEXT I
```

PLAY Statement (ON, OFF and STOP)

(BASICA only)

Syntax: **PLAY ON**

PLAY OFF

PLAY STOP

Purpose: Enables and disables trapping of a specified size of the music buffer in a BASIC program.

Comments: PLAY ON enables a ON PLAY statement activity. After PLAY ON statement, if a non-zero line number was included in the ON PLAY statement then whenever BASIC starts a new statement it checks to see if the music buffer has gone to less than n notes. If so, it will perform a GOSUB to the specified line number.

PLAY OFF disables trapping; if a PLAY activity takes place, the event is not noted.

After PLAY STOP, no trapping takes place, but if a PLAY activity takes place, the action is remembered, and an ON PLAY statement will be executed as soon as trapping is enabled.

PLAY Function

(BASICA only)

Syntax: *v*=**PLAY**(*n*)

Purpose: Returns number of notes currently residing in the music background buffer.

Comments: *n* is a dummy argument of any value.

If the program is running in Music Foreground mode, PLAY(*n*) returns 0. Since the music buffer can hold up to 47 notes, 47 is the maximum value that can be returned.

You must be in the Music Background mode to return a value with PLAY(*n*).

Example:

```
100 'when 6 notes remain in the MB
200 'buffer, go to line 900
300 'and play another song for me
400 PLAY "MB ABCDEFG"
500 IF PLAY(0)=6 GOTO 900
•
•
•
900 PLAY "MB 05 T165 L8 MS CC# DD#"
```

PMAP Function

(BASICA only)

Syntax: *v*=**PMAP**(*x*,*n*)

Purpose: Converts logical coordinates to physical coordinates and physical coordinates to logical coordinates. Graphics mode only

Comments: *x* is the coordinate of the point to be mapped.

n is a number ranging from 0 to 3 thats maps according to the following table:

0 maps the *x* logical coordinate to the *x* physical coordinate

1 maps the *y* logical coordinate to the *y* physical coordinate

2 maps the *x* physical coordinate to the *x* logical coordinate

3 maps the *y* physical coordinate to the *y* logical coordinate

PMAP translate coordinates between the logical system, defined by the WINDOW and VIEW, to the physical system. Physical coordinates are the coordinates of the screen, beginning with (0,0) in the upper left hand corner, and (639,199) (high resolution) or (319,199) (medium resolution).

To map logical coordinate values to the physical coordinate system, use PMAP (x,0) and PMAP (x,1).

To map physical coordinate values to the logical coordinate system, use PMAP (x,2) and PMAP (x,3).

Examples: The following example maps the logical coordinates (20,20) to their corresponding physical points on the screen.

```
10 SCREEN 2
20 WINDOW (10,10)–(50,50)
30 X=PMAP(20,0):Y=PMAP(20,1)
40 PRINT "PHYSICAL X=";X
50 PRINT "PHYSICAL Y=";Y
RUN
PHYSICAL X= 160
PHYSICAL Y= 149
Ok
```

POINT Function

Syntax: *v* = **POINT**(*x,y*)
v = **POINT**(*n*)

Purpose: Returns the color of a specified screen point or current coordinate of the last referenced point.

Comments: (*x,y*) represent the coordinates of the point to be used, which must be stated in absolute form.

−1 is returned if the specified point is out of range. In medium resolution, the valid returns are zero, 1, 2, or 3. In high resolution, there are two valid returns: 0 and 1.

POINT (*n*) returns the value of the current *x* or *y* coordinate of the last referenced point (Advanced BASIC 2.0 only). The value of *n* can be 0, 1, 2, or 3.

0	the current physical *x* coordinate is returned.
1	the current physical *y* coordinate is returned.
2	the current logical *x* coordinate is returned.
3	the current logical *y* coordinate is returned.

For more information, see "WINDOW Statement" in this chapter.

Examples: The following statements will invert the current setting of point (M,M).

```
50 SCREEN 2
100 IF POINT (M,M)< >0 THEN PRESET(M,M)
                      ELSE PSET (M,M)
```

or

```
100 PSET(M,M),1-POINT(M,M)
```

The following example illustrates the use of the POINT function in BASIC 2.0.

```
100 CLS:SCREEN 1,0
200 PRINT "POINT but WINDOW inactive"
300 GOSUB 1100
400 WINDOW (–10,–10)–(10,10)
500 PRINT "POINT and WINDOW active"
600 GOSUB 1100
700 END
1100 PSET (5,5)
1200 FOR J=0 TO 3
1300 PRINT POINT(J);
1400 NEXT
1500 PRINT
1600 RETURN
```

POKE Statement

Syntax: **POKE** *n,m*

Purpose: Writes one byte into a specified memory location.

Comments: *n* is a number from zero to 65535 that indicates the address in memory where the data is to be written, and represents an offset from the current segment as defined by the latest DEF SEG statement.

m represents the data that will be written to the specified address, and must be from zero to 255.

POKE is the complementary statement to the PEEK function, which is used to read a single byte. Both POKE and PEEK can be helpful for storing data efficiently, as well as loading, and passing arguments and results to and from, machine language subroutines.

Warning: BASIC performs no checking on the address. So take care to avoid the memory areas containing BASIC's stack and variable area, or your BASIC program.

Examples: The following statement sets the foreground color in graphic mode to 2.

```
100 DEF SEG
110 POKE &H4E,2
```

POS Function

Syntax: *v* =**POS**(*n*)

Purpose: Returns the current column position of the cursor.

Comments: *n* is entered as a "dummy" argument.

POS returns the current horizontal (column) cursor location. To return the current line position of the cursor, use the CSRLIN function.

Examples:

```
IF POS(0)>65 THEN BEEP
```

This example beeps (moves the cursor to the start of the next line) if the cursor has moved horizontally beyond position 65 on the screen.

PRINT Statement

Syntax: **PRINT** [*list of expressions*] [;]

? [*list of expressions*] [;]

Purpose: Displays specified data on the screen.

Comments: *list of expressions* is a list of numeric and/or string expressions that are separated by either commas, blanks, or semicolons. Any listed string constants must be enclosed in quotes.

If no *list of expressions* is given, a blank line will be displayed. When a *list of expressions* is included, the values of the expressions are displayed.

A question mark can be used as a short way of entering PRINT when the BASIC program editor is in use.

Print Positions

The position of each displayed item is determined by the punctuation separating the listed items. BASIC divides the output line into print zones, each 14 spaces long. In the *list of expressions,* a comma causes the next value to be displayed at the beginning of the following zone. A semicolon causes the next value to be printed immediately following the last value. One or more blanks between expressions has the same effect as a semicolon.

If the *list of expressions* is ended by a comma, semicolon, or SPC or TAB function, the next PRINT statement will begin printing on the same line, spacing accordingly. If none of these ends the list, a carriage return will be printed at the end of the line, that is, the cursor will move to the beginning of the next line.

When the length of the value to be printed is more than the number of character positions remaining on the current line, the value will be printed at the beginning of the following line. If the value is longer than the defined screen WIDTH, BASIC prints as much as possible on the current line, then continues printing the remainder on the next screen line.

Displayed numbers are always followed by a space. Positive numbers are preceded by a space and negative numbers by a minus sign. If single-precision numbers can be represented by 7 or fewer digits in fixed point format no less accurately than in floating point format, they are output in fixed point or integer format. For example, 10^(-7) is displayed .0000001 and 10^(-8) as 1E-8.

The LPRINT statement is used to print on a printer.

Examples: In the first example, the commas used to separate the PRINT statement expressions cause each value to be printed at the beginning of the next print zone.

```
100 X=2
200 PRINT X+10, X-2, X*(-5)
300 END
RUN
 12            0            -10
Ok
```

COMMANDS STATEMENTS

In the next example, the semicolon at the end of line 200 causes both PRINT statements to be executed on the same line.

```
100 INPUT J
200 PRINT J "SQUARED IS" J^2 "AND";
300 PRINT J "CUBED IS" J^3
400 PRINT
500 GOTO 100
RUN
? 3
 3 SQUARED IS 9 AND 3 CUBED IS 27.

? 27
 27 SQUARED IS 729 AND 27 CUBED IS 19683

?
```

In the last example, the semicolons in the PRINT statement cause each value to be printed immediately after the last. Two spaces separate each number because a printed number is followed by a space, and positive numbers preceded by a space. In line 400, the question mark replaces the word PRINT.

```
100 FOR I = 1 TO 5
200 X=X+2
300 Y=Y+4
400 ?X;Y;
500 NEXT I
RUN
 2 4 4 8 6 12 8 16 10 20
Ok
```

PRINT USING Statement

Syntax: **PRINT USING** *v$; list of expressions* [;]

Purpose: Prints data in a specified format.

Comments: *v$* is a string constant or variable consisting of special formatting characters that specify the field and the format of the printed data.

list of expressions specifies the string or numeric expressions that are to be printed, which must be separated by semicolons or commas.

Printing Strings: When printing strings, one of three formatting characters is used to specify the format of the string field:

!	Only the first character of the given string will be printed.
\n spaces\	2+n characters of the string will be printed. If the backslashes appear with no spaces, two characters will be printed; if they are separated by one space, three characters will be printed, etc.

When the string is longer than the output field, extra characters are ignored. When the field is longer than the string, the string is left-justified in the field and spaces are inserted to its right. For example:

```
100 M$="TAKE": N$="OUT"
300 PRINT USING "!";M$;N$
400 PRINT USING "\  \";MS;N$
500 PRINT USING "\    \";M$;N$;"!!"
RUN
TO
TAKEOUT
TAKE OUT !!
```

& A variable length string field will be output exactly as input. For example:

```
100 M$="TAKE": N$="OUT"
200 PRINT USING "!"; M$;
300 PRINT USING "&"; N$
RUN
TOUT
```

Printing Numbers: When printing numbers, the following special characters are used to specify the format of the numeric field:

A number sign represents digit positions, each of which is always filled. If the number to be printed has less digits than specified positions, then it is right-justified (preceded by spaces) in the filed.

A decimal point may be placed at any position in the field. If the format string specifies that the decimal point is to be preceded by a digit, that digit will always be printed (as zero if necessary). Numbers are rounded as required. Several examples follow:

```
PRINT USING "##.##";.55
 0.55

PRINT USING "###.##";310.654
310.65

PRINT USING "##.##   ";10.1,5.3,66.779,.234
10.10  5.30  66.78  0.23
```

In the last example, three spaces are inserted at the end of the format string to separate the printed numbers on the line.

+ A plus sign placed at the beginning of a format string specifies that the sign of the printed number (whether plus or minus) should appear before the number. A plus sign at the end of the format string means that the sign should be placed after the printed number.

− A minus sign at the end of the format field is used to print negative number with a trailing minus sign. Two example statements follow:

```
PRINT USING "+##.##   ";-38.95,1.4,59.6,-.9
-38.95  +1.40  +59.60  -0.90

PRINT USING "##.##-   ";-38.95,21.449,-7.05
38.95-     21.45     7.05-
```

** A double asterisk at the beginning of the format string specifies that leading spaces in the numeric field will be filled with asterisks.

** can also specify positions for two additional digits.

```
PRINT USING "**#.#    ";10.39,-0.5,740.1
*10.4  *-0.5   740.1
```

$$ — A double dollar sign causes a dollar sign to be printed immediately preceding the formatted number. $$ specifies two more digit positions, one of which is the dollar sign. Exponential format may not be used with $$. Negative numbers may only be used if they are printed with a trailing minus sign.

```
PRINT USING "$$###.##   ";456.78,5921.84
  $456.78  $5921.84
```

**$ — When **$ is placed at the beginning of a format string, the effects of the above two symbols are combined. Leading spaces are filled with asterisks and a dollar sign precedes the number. **$ also specifies another three digit positions, one of which is used for the dollar sign.

```
PRINT USING "**$##.##";2.35
***$2.35
```

A comma placed to the left of the decimal point in a formatting string causes a comma to precede every third digit to the left of the decimal point. A comma at the end of the format string is printed as part of the string. Such a comma can also specify another digit position.

The comma has no effect when used with the exponential (∧∧∧∧) format (see below).

```
PRINT USING "####,.##";5234.5
5,234.50

PRINT USING "####.##,";5234.5
5234.50,
```

∧∧∧∧ Four carets placed after the digit position characters specify exponential format. The carets define space for E±nn or D±nn to be printed. Any decimal point position may be indicated. Significant digits will be left-justified, and the exponent adjusted. If a leading + or trailing + or − is not specified, one digit position to the left of the decimal point is used to print a space or a minus sign.

```
PRINT USING "##.##∧∧∧∧";534.56
 5.35E+02
Ok
PRINT USING ".###∧∧∧∧-";-78888
 .789E+05-
Ok
PRINT USING "+.##∧∧∧∧";523
 +.52E+03
Ok
```

_ An underscore in the format string specifies that the next character is to be output as a literal character, that is, exactly as it appears in the format string.

COMMANDS STATEMENS

```
PRINT USING "_###.##_!";11.34
 #11.34!
```

The underscore itself may be printed by placing "__" in the format string.

If the number to be printed is larger than the specified numeric filed, % precedes the printed number. If rounding makes the number exceed the field, % precedes the rounded number.

```
PRINT USING "##.##";110.22
%110.22
Ok
PRINT USING ".## ";.9999
%1.00
Ok
```

The specified number of digits may not exceed 24 or an "Illegal function call" error will occur.

PRINT # and PRINT #USING Statements

Syntax: **PRINT** *#filenum,* [USING *v$*;] *list of exps*

Purpose: Writes data into a specified file.

Comments: *filenum* is the number assigned to the file when it was OPENed for output.

v$ is a string expression composed of formatting characters as described for the PRINT USING statement.

list of exps is a group of numeric and/or string expressions to be written to the file.

PRINT # writes an image of the data, as it would be displayed on the screen with a PRINT statement, to the file.

Numeric expressions in the list can be separated by semicolons or commas. If commas are used, the extra blanks inserted between print fields are written with the data to the file. When semicolons are used, the data is written to the file just as a PRINT statement would display it. Listed string expressions must be separated by semicolons.

When string expressions are written to the file, no delimiters are placed between the items of data.

For example, if F$="FILE" and B$="15", the statement

```
PRINT #1,F$;B$
```

would write FILE15 to the file, which cannot be read as two separate strings. To correct this, place explicit delimiters as data into the PRINT # statement as follows:

```
PRINT #1,F$;",";B$
```

The image then written to the file is:

```
FILE,15
```

which can be read back out into two strings.

When strings containing commas, semicolons, significant leading blanks, carriage returns, or line feeds are written to the file, they should be explicitly delimited by double quotes using CHR$(34), as shown below.

```
PRINT #1,CHR$(34);F$;CHR$(34);CHR$(34);
 B$;CHR$(34)
```

The following image would be written to the file:

```
"FILE, PERSONNEL""  15"
```

Delimiter problems can be avoided by using the WRITE # statement which enters delimiters automatically.

COMMANDS STATEMENS

PSET and PRESET Statement

Syntax: **PSET** (*x,y*) [*,color*]

PRESET (*x,y*) [*,color*]

Purpose: Draws a point at a specified screen position. Graphics mode only.

Comments: (*x,y*) represent the coordinates of the point to be set, and may be in absolute or relative form.

color specifies the color of the point, and may be from zero to 3. In medium resolution, the color is selected from the current palette as defined by the COLOR statement. Zero is the background color, and the default is the foreground color, number 3. In high resolution, if *color* zero, black is used, and the default, 1, indicates white.

The only difference between these statements is that if *color* is not included with PRESET, the background color zero is selected.

If an out-of-range coordinate is included with PSET or PRESET no action is taken nor does an error occur.

Examples: Lines 200 through 400 of this example draw a diagonal line from the point (0,0) to the point (100,100). Then lines 600 through 800 erase the line by setting each point to color zero.

```
100 SCREEN 1
200 FOR J=0 TO 100
300 PSET (J,J)
400 NEXT
500 'erase the line
600 FOR J=0 TO 100
700 PRESET STEP (-1, -1)
800 NEXT
```

PUT Statement (Files)

Syntax: **PUT** [#]*filenum* [,*number*]

Purpose: Writes a record from a random file buffer to the file.

Comments: *filenum* represents the number under which the file was OPENed.

number is the record number of the record to be written, and can be from 1 to 32767.

If *number* is omitted, the default is the next available record number (after the latest PUT).

PRINT #, PRINT # USING, WRITE #, LSET, and RSET are used to place characters in the buffer prior to a PUT statement. With WRITE #, the buffer is padded with spaces up to the carriage return.

An attempt to read or write past the end of the buffer will cause a "Filed overflow" error.

PUT can be applied to a communications file, in which case *number* represents the number of bytes to write to the file. This number must be less than or equal to that set by the OPEN"COM... statement LEN option.

PUT Statement (Graphics)

(BASICA only)

Syntax: **PUT** (*x,y*) *,array* [*,action*]

Purpose: Writes colors on a specified screen area.

Comments: (*x,y*) represents the coordinates of the top left corner of the image to be transferred.

array is the name of numeric array containing the information to be transferred. The description of the Graphics GET statement gives more information on this array.

action is one of the following: PSET, PRESET, XOR, OR, and AND, with XOR the default.

PUT performs the opposite function of GET in the sense that it takes data from an array and puts it on the screen. However, it also gives the option of interacting with the data transferred to the screen, by using the action parameter.

PSET as a PUT action simply places the data from the array on the screen, so it is the true opposite of GET.

PRESET acts the same as PSET except that a negative image is displayed. That is, a color of zero in the array causes the corresponding point to be color number 3, and vice versa; and a value of 1 in the array makes the corresponding point color number 2, and vice versa.

AND transfers the image only if an image already exists under the transferred image.

OR superimposes the array image onto an existing image.

XOR is a special mode that can be used for animation. It inverts the screen points where a point exists in the array image. A unique property of XOR is especially useful for animation: when an image is PUT against a complex background *twice*, the background is restored unchanged, allowing you to move an object without obliterating the background.

In medium resolution, AND, XOR, and OR effect on color as shown below:

AND

array value

screen	0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

COMMANDS STATEMENS

OR

array value

screen	0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

XOR

array value

screen	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

An object can be animated as follows:

1. PUT the object on the screen (using XOR).
2. Recalculate the object's new position.
3. PUT the object on the screen (using XOR) a second time in the old location, to remove the old image.

4. Return to step 1, this time PUTting the object in the new location.

When movement is done this way, the background is left unchanged. Flicker can be reduced by shortening the time between steps 4 and 1, and making sure there is enough of a time delay between steps 1 and 3. If more than one object is being animated, all should be processed at once, one step at a time.

If preserving the background is not important, animation can be done with the PSET action verb. But remember to have an image area to contain the "before" and "after" images of the object, so that the extra area will effectively erase the old image. This method may be somewhat faster than the method with XOR described above, as only one PUT is required to move an object (although you must PUT a larger image).

If the image to be transferred is too large to fit on the screen, you will get an "Illegal function call" error message.

RANDOMIZE Statement

Syntax: **RANDOMIZE** [*n*]
RANDOMIZE TIMER

Purpose: Reseeds the random number generator.

Comments: *n* represents an integer, single- or double-precision expression to be used as the random number seed.

If *n* is omitted, execution is suspended and the following prompt appears:

```
Random number seed (-32768 to 32767)?
```

before RANDOMIZE is executed.

If the random number generator is not reseeded, the RND function will return the same sequence of random numbers each time the program is run. To change the sequence whenever the program is run, place a RANDOMIZE statement at the beginning of the program and change the seed with each run.

BASIC 2.0 and later provide a way to reseed the number generator without prompting the user. Use the TIMER function to get a new random number seed each time the program runs.

Examples: The following statements will cause the prompt for the random number seed to be displayed four times.

```
100 RANDOMIZE
200 FOR J=1 TO 4
300 PRINT RND;
400 NEXT J
RUN
Random number seed (-32768 to 32767)?
```

If you enter 3, the program will print:

```
    .2226007     .5941419     .2414202     .2013798
```

In the next example, the rightmost digit of current time returned by TIME$ is used as the seed.

```
100 N=VAL(RIGHT$(TIME$,1))
200 RANDOMIZE (N)
300 FOR J=1 TO 4
400 PRINT RND;
500 NEXT
RUN
 .4417627  .1085309  .182628  .9246312
Ok
```

In this example of the TIMER function, notice that a different sequence of numbers appears each time the program is run.

```
100 RANDOMIZE TIMER
200 FOR J=1 TO 4
300 PRINT RND;
400 NEXT
RUN
.9004419 .4199934 .1857408 .1027928
Ok
RUN
.251745 .8504591 .4412188 .6957341
Ok
```

COMMANDS STATEMENS

READ Statement

Syntax: **READ** *variable* [,*variable*]...

Purpose: Reads values from a DATA statement and assigns each to a variable.

Comments: *variable* is the name of a numeric or string variable or array element to which a value from a DATA table is to be assigned.

A READ statement assigns a value from a DATA statement to a variables in the READ statement. The READ statement variable(s) can be numeric or string and the DATA value(s) must correspond in type with the associated variable(s). If this is not the case, a "Syntax error" occurs.

A single READ statement may access any number of DATA statement, which are read in the order they appear, or several READ statements may access the same DATA statement. If the number of READ variables exceeds the number of DATA elements, an "Out of data" error occurs. If the number of READ variables is less than the number of DATA elements, subsequent READ statements start reading data at the first unread element. If no more READ statements appear, the extra data is ignored.

The RESTORE statement is used to reread data from any DATA statement line.

Examples:

```
•
•
•
80 FOR I=1 to 10
90 READ X(I)
100 NEXT I
110 DATA 2.08, 5.19, 3.12, 3.98, 4.24
120 DATA 5.08, 5.55, 4.00, 3.16, 3.37
•
•
•
```

These statements read the DATA statement values into the array X. After execution of the READ loop, the value of X(1) is 2.08, X(2) is 5.19, etc.

```
100 PRINT "CITY", "STATE", " ZIPCODE"
200 READ CTY$,STA$,ZIP
300 DATA "DENVER,",COLORADO,80211
400 PRINT CTY$,STA$,ZIP
RUN
CITY        STATE       ZIPCODE
DENVER,     COLORADO    80211
Ok
```

This program reads both string and numeric data from the line 300 DATA statement. Note that quotes are not needed around COLORADO, because it doesn't contain commas, semicolons, or significant leading or trailing blanks. But quotes are needed around "DENVER," because it includes a comma. (The comma after COLORADO in the DATA statement is the delimiter between DATA fields.)

REM Statement

Syntax: **REM** *remark*

Purpose: Inserts comments into a program.

Comments: *remark* can be any sequence of characters.

REM statements are non-executable but are included in any program listing. However, they do show execution time somewhat, and use memory space.

REM statements may be branched to from a GOTO or GOSUB statement, and execution will continue with the next executable statement after the REM.

Remarks preceded by a single quote or :REM may be added at the end of a line.

Examples:

```
100 REM calculating average velocity
200 NUM=0: REM set NUM to zero
300 FOR J=1 to 20
400 NUM=NUM+V(J)
•
•
•
```

or

```
100 REM calculating average velocity
200 NUM=0 ' set NUM to zero
300 FOR J=1 TO 20
400 NUM=NUM+V(J)
•
•
•
```

RENUM Command

Syntax: **RENUM** [*newnum*] [,[*oldnum*] [,*increment*]]

Purpose: Renumbers specified program lines.

Comments: *newnum* represents the first line number that will be used in the new numbering sequence, which defaults to 10 if omitted.

oldnum represents the current program line where renumbering is to start, which defaults to line 1 if omitted.

increment is the amount added to each new line number to produce the next, which defaults to 10.

In the RENUMbering process, all line number references following GOTO, GOSUB, THEN, ELSE, ON...GOTO, ON...GOSUB, RESTORE, RESUME, and ERL test statements are changed to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Underfined line number xxxxx in yyyyy" is displayed. The incorrect line number reference (xxxxx) is not changed, but line number yyyyy can be RENUMbered.

RENUM does not change the line sequence, only that of the line numbers, or will it create line numbers greater than 65529.

Examples:

```
RENUM
```

Renumbers all the program lines, beginning with 10 and incrementing the numbers by 10.

```
RENUM 300,,50
```

Renumbers the entire program, starting with line 300 and incrementing by 50.

```
RENUM 1000,900,20
```

Renumbers the lines from 900 up so they start with line number 1000 and are incremented by 20.

RESET Command

Syntax: **RESET**

Purpose: Closes all open disk files and clears the system buffer.

Comments: If all open files are on disk, then RESET is the same as CLOSE without file numbers.

RESTORE Statement

Syntax: **RESTORE** [*line*]

Purpose: Allows rereading of a specified DATA statement.

Comments: *line* represents the line number of a DATA statement in the program.

A RESTORE statement without a line number causes the next READ statement to access the first item in the first DATA statement in the program. If *line* is specified, the next READ statement will access the first item in the specified DATA statement.

Examples: The RESTORE statement in the program below causes the DATA statement to be read twice.

```
100 READ X,Y,Z
200 RESTORE
300 READ A,B,C
400 DATA 57,68,90
500 PRINT X;Y;Z
600 PRINT A;B;C
RUN
  57  68  90
  57  68  90
Ok
```

RESUME Statement

Syntax: **RESUME** [0]

RESUME NEXT

RESUME *line*

Purpose: Continues program execution after an error recovery procedure has been performed.

Comments: The given formats have the following effects:

RESUME or RESUME 0 causes execution to continue at the statement that caused the error.

Renumbering a program containing a RESUME 0 statement causes an "Undefined line number" error. The statement will still read RESUME 0, which will be correct.

RESUME NEXT causes execution to resume at the statement immediately following the one that caused the error.

RESUME *line* causes execution to resume at the specified line number.

A RESUME statement that is not positioned inside an error trap routine causes a "RESUME without error" message to be displayed.

Examples: Line 100 in the following example states that if errors occur, the program should branch to the error-handling routine beginning on line 1000. The error routine tests the nature and location of the error, and directs control back to line 800 if error 300 occurred on line 900.

```
100 ON ERROR GOTO 1000
•
•
•
1000 IF (ERR=300) AND (ERL=900) THEN
     PRINT"TRY AGAIN":RESUME 800
```

COMMANDS STATEMENS

RETURN Statement

Syntax: **RETURN** [*line*]

Purpose: Specifies where execution should resume after a subroutine.

Comments: *line* represents the number of the program line where execution should continue.

RETURN *line* is used to direct control after a subroutine, and also permits non-local returns from event-trapping routines. From one of these routines you will often want to return to the program at a fixed line number while still eliminating the GOSUB entry created by trap. Use of the non-local RETURN must be done carefully, however, since WHILEs, FORs, or other GOSUBs active at the time of the trap remain active.

Refer to GOSUB and RETURN statement.

RIGHT$ Function

Syntax: *v$*=**RIGHT$**(*x$*,*n*)

Purpose: Returns the rightmost *n* characters of a given string.

Comments: *x$* can be any string expression.

n represents an integer expression that specifies the number of characters to be returned as the result.

If *n* is greater than or equal to the length of *x$*, then *x$* will be returned. If *n* is zero, the null string (zero length) will be returned.

Refer also to the MID$ and LEFT$ functions.

Examples: The last five characters of the string A$ are returned in the following example:

```
100 A$="DISK BASIC"
200 PRINT RIGHT$(A$,5)
RUN
BASIC
Ok
```

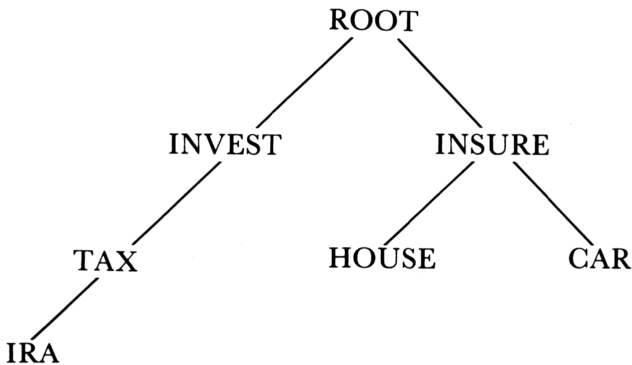
RMDIR Statement

Syntax: **RMDIR** *path*

Purpose: Removes a directory.

Comments: *path* is a valid string expression naming the subdirectory to be removed from the existing directory. The string must not exceed 63 characters.

Examples:



These examples are taken from the directory illustrated on the previous page.

Starting in the root directory, remove the directory called IRA.

```
RMDIR "INVEST\TAX\IRA"
```

Make INSURE the current directory and remove the directory called CAR.

```
CHDIR "INSURE"
RMDIR "CAR"
```

You can accomplish the same end result as the above example by making the root the current directory and then removing CAR.

```
CHDIR "\"
RMDIR "INSURE\CAR"
```

You cannot remove the directory preceding the current directory. If TAX were the current directory and you try to remove the INVEST directory, you get a **Path/file access error** message. "File not found" appears if you try to use the KILL command to remove a directory.

RND Function

Syntax: *v*=**RND**[(*x*)]

Purpose: Returns a random number between zero and 1.

Comments: *x* is a numeric expression that affects the returned value as follow:

The same random number sequence is generated whenever a program is run unles the random number generator is reseeded. This is usually done with the RANDOMIZE statement. The generator can also be reseeded by calling the RND function with a negative argument, which always generates the particular sequence for the given *x*. This sequence is not affected by RANDOMIZE, so using a different negative *x* for each run generates a different sequence each time.

If *x* is positive or omitted, the next random number in the sequence is generated.

If *x* is zero, the last generated number is repeated.

To generate random numbers from zero through *n*, use the formula:

INT(RND*(*n*+1))

Examples: This example generates ten random numbers between zero and ten.

```
10 X=RND(-1)
20 FOR I=1 TO 10
30 PRINT INT (RND(1)*11);
40 NEXT
RUN
 7 5 0 1 6 1 6 3 0 4
Ok
```

RUN Command

Syntax: **RUN** [*line*]

RUN *filespec*[,R]

Purpose: Executes the program in memory.

Comments: *line* represents the line number of the program in memory where execution is to begin.
filespec represents the string expression for the file specification.

RUN and RUN *line* begin execution of the program in memory. If *line* is included, execution starts with the specified line number. RUN alone begins execution at the lowest line number.

RUN *filespec* causes all open files to be closed, and the current contents of memory to be deleted; then the specified file is loaded into memory and run. However, if R is included, all data files will remain open.

Execution of a RUN command turns off any sound that is running and resets to Music Foreground. Also, STRIG is reset to OFF.

Examples:

```
10 PRINT 10^3
RUN
 1000
Ok
100 A=10:B=50
200 PRINT B-A
RUN 200
 0
Ok
```

In this example, the first form of RUN is used on two very tiny programs. The first runs from the beginning. The RUN *line* option is used in the second example, so the program runs from line 200. Thus, line 100 is not executed, so A and B are not assigned, and zero is printed because all numeric variables have an initial value of zero.

```
RUN "PROG2",R
```

The next example loads the program "PROG2" from disk and runs it, leaving files open.

SAVE Command

Syntax: **SAVE** *filespec* [,A]

SAVE *filespec* [,P]

Purpose: Stores a BASIC program on disk.

Comments: *filespec* represents the string expression for the file specification.

If the filename is eight characters or less and no extension is given, the extension .BAS is supplied by BASIC. A disk file with the same filename will be written over.

When A is included, the program is saved in ASCII format. Otherwise, BASIC uses a compressed binary format. ASCII files use more storage space, but some types of access require that files be in this format. For instance, a file to be MERGed must be stored in ASCII format. A program saved in ASCII can be read as a data file.

The P (protection) option will save the program in an encoded binary format. A protected program cannot be LISTed or EDITed and cannot be "unprotected".

Examples: The program in memory is saved as a disk file named PROG2.BAS:

```
SAVE "PROG2"
```

PROG2.BAS is saved in ASCII, so it may later be MERGed:

```
SAVE "PROG2",A
```

PROG2.SEC is saved into Drive B, protected so it cannot be altered:

```
SAVE "B:PROG2.SEC",P
```

SCREEN Function

Syntax: *v*=**SCREEN**(*row,col*[,*z*])

Purpose: Returns either the ASCII code (zero–255) for a specified screen character, or its color. ASCII codes are listed in Appendix C.

Comments: *row* is a numeric expression whose value ranges from 1 to 25.

col is a numeric expression whose value ranges from 1 to either 40 or 80, depending on the WIDTH setting.

Row and *col* are used to specify the position of a point on the screen.

z is a numeric expression whose result is a true or false value, and is valid only in text mode.

If *z* is included and is true (non-zero), the return, though still a number from zero to 255, represents the color attribute for the character rather than the ASCII code. This number, *v*, may be interpreted as follows:

(*v* MOD 16) represents the foreground color.

(((*v-foreground*)/16) MOD 128) represents the background color, with *foreground* calculated as above.

(*v*>127) is true (−1) for a blinking character, otherwise false (zero).

The COLOR statement gives a list of colors and their associated numbers.

In graphics mode, if the specified location contains graphic information (points or lines, as opposed to just a character), then zero is returned.

Examples: The ASCII code for the character as position (70,70) will be assigned to J.

```
10 J=SCREEN (70,70)
```

Here the color attribute of the character in the upper left hand corner of the screen will be assigned to J.

```
100 J=SCREEN (1,1,1)
```

SCREEN Statement

Syntax: **SCREEN** [*mode*] [,[*burst*] [,[*apage*] [,*vpage*]]]

Purpose: Sets screen attributes that control the subsequent display.

Comments: *mode* represents a numeric expression that results in an integer value of zero, 1, or 2. Valid vaules are:

zero=Text mode with current width (40 or 80).

1=Medium resolution graphics mode (300×200)

2=High resolution graphics mode (640×200)

burst is a numeric expression that results in a true or false value. In medium resolution graphics mode (*mode*=1), a true value disables color, and a false falue enables it. This parameter cannot affect high resolution (*mode*=2) and text mode (*mode*=zero).

apage (active page)
is an integer expression ranging from zero to 7 for width 40, or zero to 3 for width 80 that selects the page to be written to by output statements to the screen, and is valid only in text mode.

vpage (visual page)
specifies which page is to be displayed on the screen, in the same way as *apage*, which may be different. *Vpage* is valid only in text mode, and if not specified, defaults to *apage*.

If all parameters are valid, the new screen mode is stored, the display is erased, the foreground color is set to white, and the background and border colors to black. Any subsequent screen output will be displayed according to the newly stored screen attributes. If the new mode is the same as the previous one, nothing changes.

In text mode, with only *apage* and *vpage* specified, the effect is that of changing display pages for viewing. Initially, both pages default to zero. By manipulating the pages, you can display one page while building another, then switch visual pages instantaneously.

> **Note:** One cursor is shared between all the pages. To switch active pages back and forth, first save the cursor position on the current active page with POS(0) and CSRLIN before changing to another active page. Then, on returning to the original page, the cursor position can be restored with the LOCATE statement.

Any omitted parameter, except *vpage*, assumes the old value.

In a program intended to run on a machine that may have either adapter, it is recommended that you use SCREEN 0,0,0 and WIDTH 40 statements at the beginning.

Examples: The first example specifies text mode with color, and sets both the active and visual pages to zero:

```
SCREEN 0,1,0,0
```

The next statement leaves mode and color burst unchanged, but sets the active page to 1 and the visual page to 2:

```
SCREEN ,,1,2
```

This statement changes to high resolution graphics mode:

```
SCREEN 2
```

SGN Function

Syntax: *v*=**SGN**(*x*)

Purpose: Returns the sign of a specified number.

Comments: *x* represents any numeric expression.

If x is positive, 1 is returned. If *x* is zero, zero is returned. If *x* is negative, −1 is returned.

Examples: The following statement shows how SGN can be used.

```
ON SGN(J)+2 GOTO 1000,2000,3000
```

The program branches to line 1000 if J is negative, line 2000 if J is zero, and to line 3000 if J is positive.

SHELL Statement

Syntax: **SHELL** [*command string*]

Purpose: Loads and executes other programs from BASIC.

Comments: *command string* is a valid string expression containing the name of the program to run and optional command arguments. The specified program name may have any extension. If no extension is given, COMMAND will look for a .COM file, then a .EXE file, and finally a .BAT file. If the program is still not found a "Bad command or filename" error is returned.

BASIC remains in memory while the child process is running and continues when the child finishes. You cannot SHELL another copy of BASIC. If you attempt to do this, you will receive this error message "You can not run BASIC as a child of BASIC.

When the *command string* is omitted, SHELL gives you a new COMMAND shell. You may then perform any actions that COMMAND allows. To return to BASIC, type the DOS command EXIT.

A Child process that alters any open file in the BASIC parent may bring unexpected or disasterous results. When you must update such files, CLOSE them in the parent before executing the SHELL command, then re-OPEN them when the program returns to BASIC.

Before BASIC "SHELLs" to COMMAND it attempts to free any available memory, except when a /M option has been included in the BASIC command line. When /M is in effect, BASIC assumes that you intended to load data in the top of BASIC's memory block. Therefore, BASIC is prevented from compressing the workspace and an "Out of memory" error message may result. To prevent this, load machine language subroutines BEFORE BASIC is run. Place "Pocket Code" at the end of machine language subroutines that allows them to exit to DOS and stay resident. "Load" these subroutines by running them previous to BASIC.

A child process cannot 'terminate and stay resident". This does not leave BASIC enough memory to expand the workspace to its original size. If this occurs you will receive the error message "SHELL can't continue", all files are closed, and BASIC exits to DOS.

Notes:

There are several factors to keep in mind when running a Child process. BASIC is not completely protected when a SHELL statement is executed. Keep in mind the following guidelines to prevent child processes from harming the BASIC environment.

It is recommended that the state of all hardware be maintained during a SHELL command. While the implementation interface provides a means of accomplishing this, it may be necessary for BASIC users to avoid using certain devices within child processes executed by SHELL. In particular keep in mind:

child processes may modify screen mode parameters

remember to save and restore interrupt vectors used by the child

many devices placed in a specific state by BASIC may be utilized by the child. You should be aware that there may be limitations on the use of these devices (Interrupt Controller, Counter Timers, DMA Controller, I/O Latch, and Uarts).

Examples: The following example gets a new COMMAND shell, lets the user look at the directory to see files and then returns control to BASIC:

```
SHELL 'get a new COMMAND
A> DIR (user locks at directory)
A> EXIT (user exits to system)
```

The following example writes some data, sorts the data with the SHELL SORT, and reads the sorted data to write a report.

```
10 OPEN "SORTIN. DAT" FOR OUTPUT AS #1
        • 'write data to be sorted
        •
1000 CLOSE 1
1010 SHELL "SORT <SORTIN. DAT
>SORTOUT.DAT"
1020 OPEN "SORTOUT.DAT" FOR INPUT AS #1
        • 'process the sorted data
        •
```

COMMANDS STATEMENS

SIN Function

Syntax: *v*=**SIN**(*x*)

Purpose: Calculates the trigonometric sine function for a given angle.

Comments: *x* represents an angle in radians.

Multiply degrees by 3.141593/180 to convert to radians.

The return will be a single precision value.

Examples: The following example calculates the sine of 90 degrees, after converting 90 degrees to radians.

```
100 PI=3.141593
200 PRINT SIN(PI/2)
RUN
 1
Ok
```

COMMANDS STATEMENS

SOUND Statement

Syntax: **SOUND** *freq, duration*

Purpose: Generates specified sound.

Comments: *freq* represents the desired Hertz frequency (cycles per second), and must be numeric expression whose result ranges from 19 to 32767.

duration represents the desired duration in clock ticks, which occur 18.2 times per second. *duration* must be a numeric expression whose result ranges from zero to 65535.

When a SOUND statement produces a sound, the program continues executing until another SOUND statement appears. If the *duration* of the new SOUND is zero, the current SOUND is turned off. Otherwise, the program waits until the first sound is completed before executing the new SOUND statement. Sounds can be buffered so that execution will not stop when a new SOUND statement is encountered. Refer to the MB command under the PLAY statement for details.

If no SOUND statement is being executed, SOUND *x*, 0 will have no effect.

The following table lists frequencies for each note in the two octaves on either side of middle C, which is indicated with*.

Note	Frequency	Note	Frequency
C	130.810	C*	523.250
D	146.830	D	587.330
E	164.810	E	659.260
F	174.610	F	698.460
G	196.000	G	783.990
A	220.000	A	880.000
B	246.940	B	987.770
C	261.630	C	1046.500
D	293.660	D	1174.700
E	329.630	E	1318.500
F	349.230	F	1396.900
G	392.000	G	1568.000
A	440.000	A	1760.000
B	493.880	B	1975.500

Higher (or lower) notes can be approximated by doubling (or halving) the frequency of the corresponding note in the previous (following) octave.

SOUND 32767, *duration* will create periods of silence.

The duration for each beat can be calculated by dividing the beats per minute into 1092 (the number of clock ticks in one minute).

The following table shows typical tempos in terms of clock ticks:

Tempo		Beats/ Minute	Ticks/ Beat
very slow	Larghissimo		
↓	Largo	40–60	27.3–18.2
	Larghetto	60–66	18.2–16.55
	Grave		
	Lento		
	Adagio	66–76	16.55–14.37
slow	Adagietto		
↓	Andante	76–108	14.37–10.11
medium	Andantino		
↓	Moderato	108–120	10.11–9.1
fast	Allegretto		
↓	Allegro	120–168	9.1–6.5
	Vivace		
	Veloce		
	Presto	168–208	6.5–5.25
very fast	Prestissimo		

Examples: The program below creates random notes.

```
10 FOR I=1 TO 30
20 SOUND RND*1000+19,2
30 NEXT
```

SPACE$ Function

Syntax: *v$*=**SPACE$**(*n*)

Purpose: Returns a string a given number of spaces long.

Comments: *n* must be a value from zero to 255.

Examples: The following example uses the SPACE$ function to print each J on a line preceded by J spaces. An additional space appears because BASIC inserts a space in front of positive numbers.

```
100 FOR J=1 to 4
200 M$=SPACE$(J)
300 PRINT M$;J
400 NEXT J
RUN
  1
   2
    3
     4
Ok
```

COMMANDS STATEMENS

SPC Function

Syntax: **PRINT SPC**(*n*)

Purpose: Skips a given number of horizontal spaces in a PRINT statement.

Comments: *n* must be a number from zero to 32767. If greater than the defined device width, then *n* MOD *width* is used.

SPC can only be used with PRINT, LPRINT and PRINT#.

If the SPC function appears at the end of the list of data items, then BASIC does not add a carriage return, as though SPC were followed by an implied semicolon.

Examples:

```
PRINT "MOVE" SPC(15) "OVER"
MOVE               OVER
Ok
```

This example prints two words separated by 15 spaces.

COMMANDS
STATEMENS

SQR Function

Syntax: *v*=**SQR**(*x*)

Purpose: Returns the square root of a given value.

Comments: *x* must be either zero or positive.

Examples: The following example calculates the square roots of the numbers 2, 4, 6 and 8.

```
100 FOR R=2 TO 8 STEP 2
200 PRINT R, SQR(R)
300 NEXT
RUN
  2                 1.414214
  4                 2
  6                 2.44949
  8                 2.828427

Ok
```

STICK Function

Syntax: *v*=**STICK**(*n*)

Purpose: Returns the x or y coordinates of one of two joysticks.

Comments: *n* is a numeric expression whose value ranges from zero to 3, and specifies the result as follows:

zero will return the x coordinate of joystick A.

1 will return the x coordinate of joystick A.

2 will return the x coordinate of joystick B.

3 will return the y coordinate of joystick B.

If *n* is zero, all four coordinate values are retrieved and the value for joystick A is returned. STICK(1), STICK(2), and STICK(3) do not test the joystick, but instead they get the values previously retrieved by STICK(0).

The x and y range depend on your particular joysticks.

Examples: This program takes the coordinates of joystick A 50 times and prints them.

```
100 PRINT "Joystick A"
200 PRINT "x", "y"
300 FOR M=1 TO 50
400 VALS=STICK(0)
500 A1=STICK(0): A2=STICK(1)
600 PRINT A1,A2
700 NEXT
```

STOP Statement

Syntax: **STOP**

Purpose: Stops executing a program and returns to command level.

Comments: STOP statements may appear anywhere in a program to terminate execution. When a STOP statement is encountered, BASIC displays the following message:

```
Break in nnnnn
```

nnnnn is the line number where execution stopped.

The STOP statement does not close files. (The END statement closes files.)

Execution can be resumed with a CONT command.

Examples: The following example calculates the value of NUM, then stops. While the program is stopped, you can check the value of NUM. Then you can enter CONT to resume executing at line 400.

```
100 INPUT X,Y
200 NUM=X*Y
300 STOP
400 FINAL=NUM+30: PRINT FINAL
RUN
? 3.4, 2.7
Break in 300
OK
PRINT NUM
  9.18
Ok
CONT
  39.18
Ok
```

COMMANDS STATEMENS

STR$ Function

Syntax: *v$*=**STR$**(*x*)

Purpose: Returns a string representation of the value of a given numeric expression.

Comments: *x* can be any numeric expression.

If *x* is positive, the returned string includes a leading blank (the space reserved for the plus sign). For example:

```
? STR$(521); LEN(STR$(521))
 521 4
Ok
```

STR$ is the complementary function to VAL.

Examples: This example branches to different sections of the program depending on the number of digits in an entered number. The digits are counted to converting the number to a string with STR$, then branching based on the string's length.

```
100 INPUT "ENTER A NUMBER";NUM
200 ON LEN(STR$(NUM) ) GOSUB 300,400,500,600
```

STRIG Statement and Function

Syntax: STRIG statement

STRIG ON

STRIG OFF

STRIG function:

v=**STRIG**(*n*)

Purpose: Returns the status of the joystick buttons (triggers).

Comments: STRIG ON must be executed before any STRIG(*n*) function calls can be made. After STRIG ON, each time the program starts a new statement BASIC tests whether a button has been pressed.

If STRIG is OFF, no testing is done.

n is a numeric expression whose value may range from zero to 7. It specifies characteristics of the return as follows:

n=zero will return −1 if button A1 was pressed since the last STRIG(0), and zero if not.

1 will return −1 if button A1 is currently pressed, and zero if not.

2 will return −1 if button B1 was pressed since the last STRIG(2), and zero if not.

3 will return −1 if button B1 is currently pressed, and zero if not.

4 will return −1 if button A2 was pressed since the last STRIG(4), and zero if not.

5 will return-1 if button A2 is currently pressed, and zero if not.

6 will return-1 if button B2 was pressed since the last STRIG(6), and zero if not.

7 will return-1 if button B2 is currently pressed, and zero if not.

STRIG(n) Statement

Syntax: **STRIG**(*n*) ON

STRIG(*n*) OFF

STRIG(*n*) STOP

Purpose: Activates and deactivates trapping of the joystick buttons.

Comments: *n* can be 0, 2, 4, or 6, and specifies the button to be trapped as follows:

zero=button A1
2=button B1
4=button A2
6=button B2

STRIG(*n*) ON must be executed to activate trapping by the ON STRIG(*n*) statement. After STRIG(*n*) ON, whenever the program starts a new statement, BASIC checks to see whether the specified button has been pressed.

If STRIG(*n*) OFF is executed, no testing or trapping is done. Even if the button is pressed, the event is not noted.

If STRIG(*n*) STOP is executed, no trapping is done, but if the button is pressed, the event is remembered, and an immediate trap takes place if STRIG(*n*) ON is executed.

STRING$ Function

Syntax: *v$*=**STRING$**(*n*,*m*)

*v$*=**STRING$**(*n*,*x$*)

Purpose: Returns a string of length *n* composed either of characters with ASCII code *m* or the first character of a specified string.

Comments: *n* and *m* range from zero to 255.

x$ can be any string expression.

Examples: The following example repeats an ASCII value of 45 to print a string of 5 hyphens.

```
100 S$=STRING$(5,45)
200 PRINT S$ "TABLE OF CONTENTS" S$
RUN
-----TABLE OF CONTENTS-----
Ok
```

The next example repeats the first character of the string "*...".

```
100 S$="*..."
200 X$=STRING$(5,S$)
300 PRINT X$
RUN
*****
Ok
```

SWAP Statement

Syntax: **SWAP** *variable1, variable2*

Purpose: Exchanges the values of two given variables.

Comments: *variable1, variable2* represent the names of two variables or array elements.

The two variables must be the same type or a "Type mismatch" error will occur.

Examples: In the following example, when line 300 is executed, ST1$ has the value "ME" and ST2$ has the value "YOU".

```
100 ST1$="YOU":ST2$="ME":ST3$=" FOR "
200 PRINT ST1$ ST3$ ST2$
300 SWAP ST1$, ST2$
400 PRINT ST1$ ST3$ ST2$
RUN
YOU FOR ME
ME FOR YOU
Ok
```

SYSTEM Command

Syntax: **SYSTEM**

Purpose: Exits BASIC and return to DOS.

Comments: All files are closed before the return to DOS. The BASIC program in memory will be lost.

If you entered BASIC via a Batch file from DOS, SYSTEM returns you to the Batch file, which continues executing where it left off.

TAB Function

Syntax: **PRINT TAB**(*n*)

Purpose: Skips to a given print line position to print the next item.

Comments: *n* must range from 1 to 32767.

If the current print position is already past space *n*, TAB goes to position *n* on the next line. Space 1 is at the left margin, and rightmost position is the defined WIDTH.

TAB may only be used with PRINT, LPRINT, and PRINT#

If the TAB function appears at the end of the list of data items, then BASIC will not add a carriage return, as if the TAB function had an implied semicolon after it.

Examples: TAB is used in the following example to line up the displayed information in columns.

```
100 PRINT "NAME" TAB(25) "NUMER":PRINT
200 READ M$, N$
300 PRINT M$ TAB(25) N$
400 DATA "B. J. FELDMAN","7165"
RUN
NAME                    NUMBER

B. J. FELDMAN           7165
Ok
```

COMMANDS STATEMENS

TAN Function

Syntax: v=**TAN**(x)

Purpose: Returns the trigonometric tangent of a given angle.

Comments: x represents the angle in radians. Multiply degress by 3.141593/180 to get radians.

The return is a single precision value.

Examples: This example computes the tangent of 45 degrees.

```
100 PI=3.141593
200  PRINT TAN(PI/4)
RUN
 1
Ok
```

TIMER Function

Syntax: *v*=**TIMER**

Purpose: Tracks the number of elapsed seconds since midnight or system reset. For use in BASIC 2.0 and later only.

Comments: TIMER returns a single-precision number. Fractional second are expressed in the nearest possible degree. This is a read only function.

Example:

```
100 TIME$="23:59:59"
200 FOR I=1 to 15
300 PRINT "TIME$= ";TIME$,"TIMER= ";TIMER
400 NEXT
RUN
TIME$= 23:59:59          TIMER= 86399.11
TIME$= 23:59:59          TIMER= 86399.22
TIME$= 23:59:59          TIMER= 86399.39
TIME$= 23:59:59          TIMER= 86399.5
TIME$= 23:59:59          TIMER= 86399.61
TIME$= 23:59:59          TIMER= 86399.72
TIME$= 23:59:59          TIMER= 86399.83
TIME$= 23:59:59          TIMER= 86399.94
TIME$= 23:59:59          TIMER= 86400.05
TIME$= 00:00:00          TIMER= .05
TIME$= 00:00:00          TIMER= .21
TIME$= 00:00:00          TIMER= .32
TIME$= 00:00:00          TIMER= .43
TIME$= 00:00:00          TIMER= .49
TIME$= 00:00:00          TIMER= .6
Ok
```

TIMER Statement

(BASICA only)

Syntax: **TIMER ON**

TIMER OFF

TIMER STOP

Purpose: Enables and disables trapping of the timer in a BASIC program.

Comments: TIMER ON enables timer event trapping by an ON TIMER statement. After TIMER ON statement, if a non-zero line number was included in the ON TIMER statement then whenever BASIC starts a new statement it checks to see if the specified period of time has elapsed. If so, it will perform a GOSUB to the specified line number, and BASIC starts counting again.

TIMER OFF disables trapping; if the specified period of time has elapsed, the event is not noted.

After TIMER STOP, no trapping takes place, but if the specified period of time has elapsed, the action is remembered, and an ON TIMER statement will be executed as soon as trapping is enabled.

TIME$ Variable and Statement

Syntax: TIME$ variable:

*v$* = **TIME$**

TIME$ statement:

**TIME$** = *x$*

Purpose: Sets or reads the current time.

Comments: The variable (v$=TIME$) returns the current timing of the form *hh:mm:ss,* where *hh* is the hour (00 to 23), *mm* represents minutes (00 to 59), and *ss* is seconds (00 to 59).

The statement (TIME$=x$) is used to set the current time. x$ is a string expression that indicates the time to be set, and may be specified in one of the following forms:

hh sets the hour from zero to 23, with minutes and seconds defaulting to zero.

hh:mm sets the hour and minutes. Minutes must range from zero to 59. Seconds will default to zero.

hh:mm:ss sets the hour, minutes, and seconds, with the seconds ranging from zero to 59.

Examples: The following example will display the current time.

```
PRINT TIME$
13:50:07
```

COMMANDS STATEMENS

TRON and TROFF Commands

Syntax: **TRON**

TROFF

Purpose: Traces the execution of program statements.

Comments: As a debugging aid, TRON (which can be entered in indirect mode) activates a trace flag that prints each program line number as it is executed. The numbers are printed enclosed in square brackets. TROFF turns off the trace.

Examples: In the following example, the numbers inside brackets are line numbers; the numbers not enclosed in brackets at the end of each line are the values of B, A, and C that the program prints on line 400.

```
100 A=15
200 FOR B=1 TO 2
300 C=A +10
400 PRINT B;A;C
500 A=A+10
600 NEXT
700 END
TRON
Ok
RUN
[100][200][300][400] 1  15  25
[500][600][300][400] 2  25  35
[500][600][700]
Ok
TROFF
Ok
```

USR Function

Syntax: *v* = **USR**[*n*](*arg*)

Purpose: Calls the specified machine language subroutine with a given argument.

Comments: *n* must range from zero to 9 and correspond to the digit given by the DEF USR statement for the desired routine. If *n* is omitted, a value of zero is assumed.

arg represents a numeric expression or string variable, which will be passed as the argument to the machine language subroutine.

Examples: In the following example, the address of function USR0 is specified in line 400. Line 500 calls USR0 with the argument A/2. Line 600 calls the same function again, this time with the argument A/3.

```
100 CLEAR ,&HC000
200 DEF SEG
300 BLOAD "SUB1", &HC000
400 DEF USR0=&HC000
500 X=USR0(A/2)
600 Y=USR(A/3)
```

VAL Function

Syntax: v =**VAL**($x\$$)

Purpose: Returns the numeric value of a given string.

Comments: *x$* represents a string expression.

VAL strips blanks, tabs, and line feeds from the given string to determine the result. For instance, VAL(" −3") will return −3.

If the first characters encountered in *x$* are not numeric, then VAL returns zero.

The STR$ function is used to convert numeric values to strings.

Examples: The following checks STATE by the VAL function.

```
10 READ NAM$, CITY$, STATE$, ZIP$
20 IF VAL (ZIP$)<90000 OR VAL (ZIP$)>96699
   THEN PRINT NAM$ TAB (25) "OUT OF STATE"
30 IF VAL (ZIP$)>=90801 AND VAL (ZIP$)<=90815
   THEN PRINT NAM$ TAB(25) "LONG BEACH"
```

VARPTR Function

Syntax: *v* = **VARPTR**(*variable*)

v = **VARPTR**(#*filenum*)

Purpose: Returns the memory address of a given variable or file control block.

Comments: *variable* represents the name of a numeric or string variable or array element in your program, to which a value must be assigned before the call to VARPTR.

filenum represents the number under which the file was OPENed.

The returned address is an integer from zero to 65535, which represents the offset into BASIC's Data Segment.

The first format will return the address of the first byte of data associated with *variable*.

> **Note:** Simple variables should be assigned before calling VARPTR for an array, since array addresses change whenever a new simple variable is assigned.

VARPTR is usually used to obtain the address of a variable or array that will be passed to a USR machine language subroutine. A call of the form VARPTR(A(0)) is usually made when passing an array, so that the lowest-addressed array element is returned.

The second format will return the starting address of the file control block for the specified file, which differs from the DOS file control block.

Examples: This example reads the first byte in a random file buffer.

```
100 OPEN "DATA" AS #1
200 GET #1
300 ADR1=VARPTR(#1)
400 ADR2=ADR1+188
500 ADDR%=PEEK(ADR2)
```

The following example uses VARPTR to get data from a variable. In line 300, X gets the address of the data. Integer data is stored in two bytes, beginning with the less significant byte. The actual value stored at location X is calculated in line 400. The PEEK function reads the bytes, and the second byte (containing the high-order bits) is multiplied by 256.

```
100 DEFINT A
200 A=50
300 X=VARPTR(A)
400 Y=PEEK(X) + 256*PEEK(X+1)
500 PRINT Y
```

VARPTR$ Function

Syntax: *v$*=**VARPTR$**(*variable*)

Purpose: Returns in character form the memory address of a variable, and is primarily for use with PLAY and DRAW in programs that will later be compiled.

Comments: *variable* represents the name of a program variable.

> **Note:** Simple variables should be assigned before calling VARPTR$ for an array element, since array addresses change whenever a new simple variable is assigned.

VARPTR$ returns a three-byte string in this form:

Byte 0: *type*

type is: 2 integer
3 string
4 single-precision
8 double-precision

Byte 1: low-order byte of variable address

Byte 2: high-order byte of variable address

The returned string is the same as:

CHR$(*type*)+MKI$(VARPTR(*variable*))

VARPTR$ can be used to indicate a variable name in the command string for PLAY or DRAW. For example:

```
PLAY "XA$;" or PLAY "X"+VARPTR$(A$)
PLAY "N=J;" or PLAY "N="+VARPTR$(J)
```

VIEW PRINT Statement

(BASICA only)

Syntax: **VIEW PRINT** *linenum* to *linenum*

Purpose: Sets the limits of the text window.

Comments: The first *linenum* defines the upper limits of the text window, it cannot be less than 1. The second *linenum* defines the lower limits of the text window, it cannot be more than 24.

CLS, LOCATE and SCREEN all operate within the text window defined by VIEW PRINT. The screen editor limits functions such as scrolling and cursor movement to the area within the specified boundaries.

In the text mode both CLS and <Ctrl> <Home> clear only the specified text window.

In the graphic mode, if VIEW is not specified, the text window is cleared. If VIEW is specified, CLS clears the view port and <Ctrl> <Home> clears the text window.

When no parameters are specified, the text window includes the entire screen.

Example: The text screen begins at line 5 and extends to line 15.

```
VIEW PRINT 5 TO 15
```

VIEW Statement

(BASICA only)

Syntax: **VIEW** [[**SCREEN**] [(*x*1,*y*1)–(*x*2,*y*2) [,[*color*] [,[*boundary*]]]]]

Purpose: Defines viewports (subsets of the viewing surface) onto which windows will be mapped. Graphics mode only.

Comments: (*x*1,*y*1) defines the upper-left coordinates of the viewport. (*x*2,*y*2) define the lower-right coordinates. *x* and *y* coordinates must be within the physical boundaries of the screen. Specifying coordinates outside the actual limits of the screen results in an **Illegal function call** error. Refer to "Specificying Graphics Coordinates" on page 6-25.

color allows you to fill the defined viewpoint with a color. When *color* is omitted, the viewport is not filled. In medium resolution, *color* can range from 0 to 3. 0 is background color and 3 is foreground color. In high resolution, 0 (zero) color is black, 1 is white.

boundary draws a line surrounding the viewport, if space for a boundary is available. When *boundary* is omitted, no border is drawn.

RUN or VIEW with no arguments define the entire screen as the viewport.

VIEW sorts the x and y pairs. The smallest values for x and y are place first.

VIEW (100,100)–(20,20)

becomes

VIEW (20,20)–(100,100)

$x1$ cannot equal $x2$ and $y1$ cannot equal $y2$. All other pairings of x and y are possible. The viewport must be contained within the viewing surface.

If you omit the SCREEN argument, all points plotted are relative to the viewport. This means $x1$ and $x2$ are added to the x and y coordinates prior to plotting the point on the screen. When the following example is executed, the point plotted by PSET (0,0),3 will appear at the physical screen location (10,10).

VIEW (10,10)–(200,100)

If you include the SCREEN argument, the points are absolute and may be plotted inside or outside of the viewport. Only those points within the viewports limits will be visible, however. When the following example is executed, the point plotted by PSET (0,0),3 will not appear on the screen because 0,0 is outside of the viewport. The point plotted by PSET (10,10),3 will appear and be placed in the upper-left corner as it is within the viewport.

VIEW SCREEN (10,10)–(200,100)

To scale using the VIEW statement, simply change the size of your viewport. A large viewpoint makes your objects large and small viewport makes them small.

Notes:

When using VIEW, the CLS statement clears only the current viewport. To clear the entire screen, first use VIEW to disable the viewports, then use CLS. <Ctrl> <Home> clears the text window.

Example:

```
10 SCREEN 1:CLS:KEY OFF
15 WINDOW (−1,−1)−(1,1)
20 GOSUB 1000
30 CLS
40 VIEW (10,10)−(50,50),,1
50 GOSUB 1000
60 VIEW (70,70)−(200,190),,1
70 GOSUB 1000
80 END
1000 CIRCLE (0,0),.5,1
1010 PAINT (0,0),2,1
1020 RETURN
```

WAIT Statement

Syntax: **WAIT** *port*, *n*[,*m*]

Purpose: Suspends program execution while testing the status of an input port.

Comments: *port* is the port number, and range from zero to 65535.

n, m are integer expressions whose values range from zero to 255.

The WAIT statement causes program execution to be suspended until a specified machine input port develops a specified pattern of bits.

Data read at the port is XORed with *m*, then ANDed with *n*. If the result is zero, BASIC will loop back and read the port data again. If the result is not zero, execution resumes with the next statement. If *m* is omitted, it defaults to zero.

The WAIT statement is used to test one or more bit positions on an input port. The bit position may be tested for either a 1 or a zero. The bit positions to be tested are specified by setting those positions in *n* to ones. If *m* is not specified, a 1 in any bit position in *m* (for which there is a corresponding 1-bit in *n*) causes WAIT to test for a zero for that bit.

The WAIT statement loops testing of those input bits specified by 1's in *n*. If *any one* of them is 1 (or zero if the corresponding *m* bit is 1), then the program continues on the next statement. Thus WAIT does not wait for an entire bit pattern to appear, only for one of them to occur.

Note: It is possible to enter an infinite loop with WAIT, which can be exited with Ctrl-Break or a System Reset.

Examples: To halt execution until port 30 receives a 1-bit in the third bit position:

```
500 WAIT 30,3
```

COMMANDS STATEMENS

WHILE and WEND Statements

Syntax: **WHILE** *expression*

•

•

•

(*loop statements*)

•

•

•

WEND

Purpose: Executes a series of statements as a loop for as long as a specified condition is true.

Comments: *expression* can be any numeric expression.

If the *expression* is true (not-zero), the *loop statements* are executed until the WEND statement appears. BASIC then goes back to the WHILE statement and tests the *expression*. If it is not true, execution will continue with the statement after the WEND statement.

A WHILE...WEND loop can be nested to any level. Each WEND will be matched with the most recent WHILE. An unmatched WHILE will cause a "WHILE without WEND" error, and an unmatched WEND will cause a "WEND without WHILE" error.

Examples: The following example sorts the elements of the string array X$ into alphabetical order. X$ was defined with M elements.

```
400 SWITCH=1
410 WHILE SWITCH
415    SWITCH=0
420    FOR J=1 TO M-1
430     IF X$(J)>X$(J+1) THEN
           SWAP X$(J),X$(J+1): SWITCH=1
440    NEXT J
450 WEND
```

COMMANDS STATEMENS

WIDTH Statement

Syntax: **WIDTH** *size*

WIDTH *filenum,size*

WIDTH *device,size*

Purpose: Sets the length of the output line in number of characters. After outputting the specified number of characters, BASIC will add a carriage return.

Comments: *size* is a numeric expression, whose value ranges from zero to 255 that represents the new width. WIDTH with *size*=zero is the same as WIDTH 1.

filenum is a numeric expression whose value ranges from 1 to 255 that is the number of a file opened for one of the *devices* listed below.

device is a string expression that identifies the output device. Valid *device* specifications are SCRN:, LPT1:, LPT2:, LPT3:, COM1:, or COM2:.

Depending on which device is specified, the following results are possible:

WIDTH *size*

Sets the screen line width. Either 40 or 80 is allowed for the size.

If the screen is in medium resolution graphics mode (as would result from a SCREEN 1 statement), WIDTH 80 will force the screen into high resolution (same effect as SCREEN 2).

If the screen is in high resolution graphics mode (as would result from a SCREEN 2 statement), WIDTH 40 will force the screen into medium resolution (same effect as SCREEN 1).

WIDTH *device,size*

Used as a deferred width setting for the device. This form of WIDTH stores the new width, but does not change the current width. The next OPEN to the device will use the deferred width while the file is open.

Note: LPRINT, LLIST, and LIST, "LPTn:" perform an implicit OPEN and so are affected by this statement.

WIDTH *filenum,size*

The width of the device associated with *filenum* will immediately be changed to the specified size, allowing you to change the width at will while the file is open. This form of WIDTH can be used for all devices.

Each printer's width defaults to 80 when BASIC is started. The maximum width for a line printer is 132. However, values between 132 and 255 do not cause an error.

You must set the appropriate physical width for the printer you have. Some printers are set with special codes; some with internal switches.

Line folding can be disabled by specifying a width of 255, which gives the effect of "infinite" width. This setting is the default for communications files.

Changing the width of a communications file will not alter either the receive or the transmit buffer; it will simply cause BASIC to send a carriage return character after every line defined by *size*.

Changing the screen mode affects the width only if you move between SCREEN 2 and SCREEN 1 or SCREEN 0.

Examples: In the following example, line 10 sets a printer width of 20 characters per line. Line 40 changes the width to 5 characters per line. The result is output to the printer.

```
10 OPEN "O", #1, "LPT1:"
20 WIDTH #1,20
30 PRINT#1, "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
40 WIDTH #1,5
50 PRINT #1,"123456789"
RUN
ABCDEFGHIJKLMNOPQRST
UVWXYZ
12345
6789
```

WINDOW Statement

(BASICA only)

Syntax: **WINDOW** [[**SCREEN**] (*x*1,*y*1)–(*x*2,*y*2)]

Purpose: Redefines the coordinates of the screen. Graphics mode only.

Comments: (*x*1, *y*1),(*x*2,*y*2) are logical coordinates which programmer can define. Logical coordinates are single-precision, floating-point numbers.

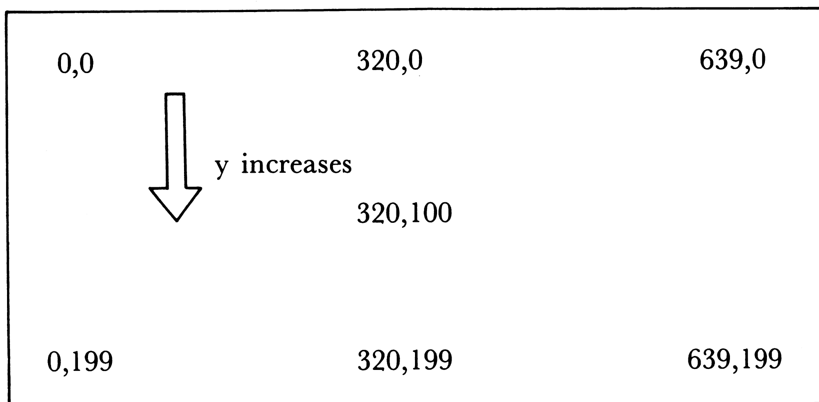
WINDOW allows you to place lines, graphs or objects using your own coordinates. The rectangular region in the logical coordinate space defined by these pairs is called a window.

These coordinates are then transformed into the appropriate physical coordinate pairs for display within the screen limits.

SCREEN, RUN or WINDOW with no arguments disables any logical coordinates. The screen returns to physical coordinates.

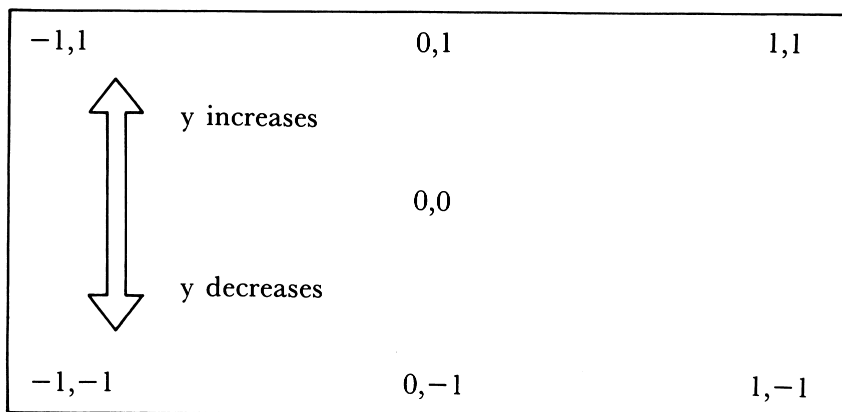
In the physical coordinate system the screen will appear with standard coordinates when you run the following:

```
NEW: SCREEN 2
```



If you omit the SCREEN agrument, WINDOW inverts the y coordinate, (x1,y1) is the lower-left coordinate and (x2,y2) is the upper right coordinate. The screen is viewed in true cartesian coordinates.

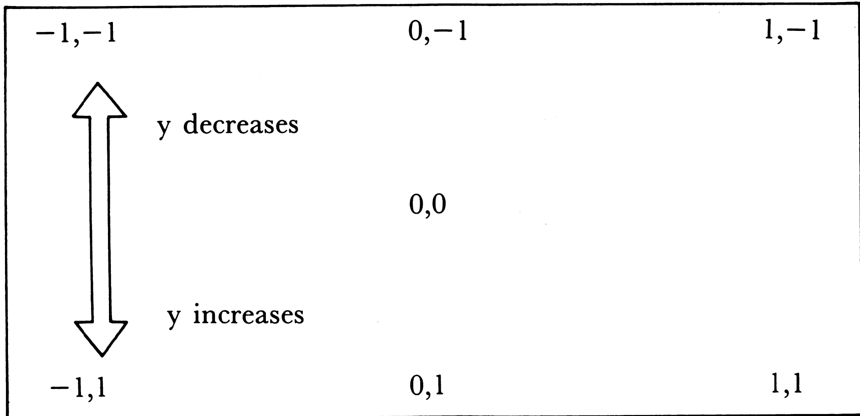
```
WINDOW (-1,-1)-(1,1)
```



COMMANDS STATEMENS

If you include the SCREEN argument, the y coordinates are not inverted, (x1,y1) is the upper-left coordinate and (x2,y2) is the lower right coordinate.

```
WINDOW SCREEN (−1, −1)−(1,1)
```



COMMANDS
STATEMENS

WINDOW sorts the x and y pairs. The smallest values for x and y are placed first.

WINDOW (100,100)–(20,20)

becomes

WINDOW (20,20)–(100,100)

*x*1 cannot equal *x*2 and *y*1 cannot equal *y*2. All other pairings of *x* and *y* are valid.

WINDOW uses line clipping, a process in which referenced points outside of a coordinate range are invisible. Objects partially within and partially without a coordinate range are cut off at the limits of the referenced range.

WINDOW allows you to "zoom" in on an object. Choosing window coordinates smaller than the images forces clipping. Only a portion of the image is displayed and magnified. You can zoom in until an object occupies the entire screen.

You may also use WINDOW to "pan". Choosing coordinates larger than an image displays the entire image, but it will be small and blank spaces appear on the sides of the screen. You can pan out until the image is just a spot on the screen.

Example:

```
100 SCREEN 1:CLS
200 WINDOW (20,20)–(10,10)
300 CIRCLE (5,5),10,1
400 REM illustration of line clipping
500 WINDOW (–50,–50)–(50,50)
600 CIRCLE (5,5),10,1 'large window, small image
```

COMMANDS STATEMENS

WRITE Statement

Syntax: **WRITE** [*list of expressions*]

Purpose: Displays specified data on the screen.

Comments: *list of expressions* represents a list of numeric and/or string expressions that are separated by commas or semicolons.

If the list of expressions is not included, a blank line is output. If the list is included, the values of the expressions are displayed.

When the values of the expressions are written, each item is separated from the previous one by a comma. Strings are enclosed in quotes. When the last item in the list is printed, BASIC adds a carriage return/line feed.

The difference between WRITE and PRINT is that WRITE places commas between the displayed items and delimits strings with quotes. Also, WRITE does not precede positive numbers with blanks.

Examples: This example illustrates how WRITE displays numeric and string values.

```
100 X=90: Y=100: C$="THAT'S IT"
200 WRITE X,Y,C$
RUN
90,100,"THAT'S IT"
Ok
```

WRITE # Statement

Syntax: **WRITE** *#filenum, list of expressions*

Purpose: Outputs data to a specified sequential file.

Comments: *filenum* represents the number under which the file was OPENed for output.

list of expressions represents a list of string and/or numeric expressions that are separated by commas or semicolons.

The difference between WRITE # and PRINT # is that WRITE # will insert commas between the items as they are written and will delimit strings with quotes. Therefore, it is not necessary for the user to explicitly delimit the listed items. Also, WRITE # does not precede a positive number with a blank. A carriage return/line feed sequence is placed after the last listed item is written.

Examples: IF ITEM$="LEAD PIPE" and NUM$="18 IN.", the statement

```
WRITE #1,ITEM$,NUM$
```

will write the following image to the file:

```
"LEAD PIPE","18 IN."
```

A later INPUT # statement, such as:

```
INPUT #1,ITEM$,NUM$
```

would input "LEAD PIPE" to ITEM$ and "18 IN." to NUM$.

COMMANDS STATEMENTS

MEMO

APPENDICES

CONTENTS

APPENDIX A—Error Messages.

APPENDIX B—Mathematical Functions.

APPENDIX C—ASCII Character Codes.

APPENDIX D—Accessing Machine Language Subroutines.

APPENDIX E—Converting a Program to Panasonic BASIC.

APPENDIX F—Executing Application Programs.

APPENDIX G—Communication I/O Procedures.

APPENDIX H—Example Programs.

APPENDIX I—Index.

MEMO

APPENDIX A
ERROR MESSAGES

INTRODUCTION A-2

ERROR MESSAGES WITH DESCRIPTIONS A-3

ALPHABETIC CROSS-REFERENCE A-19

INTRODUCTION

BASIC's error checking causes a program to stop running and display an error message when an error is encountered. These errors can be trapped and tested in a BASIC program by using the ON ERROR statement and the ERR and ERL variables. (See Chapter 7 for more information about ON ERROR, ERR, and ERL.)

This appendix includes two sections:

The "Error Messages with Descriptions" section lists and explains each BASIC error message. This section is listed numerically by error message number.

The "Alphabetic Cross-Reference" section lists each error message by its name with its associated error number.

ERROR MESSAGES WITH DESCRIPTIONS

1 NEXT without FOR

The NEXT statement lacks a corresponding FOR statement. Check to see if you used a variable in the NEXT statement that does not correspond to any previously executed and unmatched FOR statement variable.

Correct the program so the NEXT has a matching FOR.

2 Syntax error

A line contains an incorrect sequence of characters. For example, it may have an unmatched parenthesis, a misspelled command or statement, or incorrect punctuation. Another possibility is that you have a mismatch between a DATA statement and the variable type (numeric or string) in a READ statement.

When this error occurs, the BASIC program editor automatically displays the line in error, so you can immediately correct the line or the program.

3 RETURN without GOSUB

A RETURN statement requires a previous unmatched GOSUB statement.

Correct the program. If you include a STOP or END statement before the subroutine, the program will not "fall" into the subroutine code.

4 Out of DATA

A READ statement is attempting to read more data than is included in the DATA statements.

Change the program so that there are enough constants in the DATA statements for all the READ statements in the program.

5 Illegal function call

You attempted to use a parameter that was out of range. The error may also occur as the result of one of the following:

a negative or too large subscript

a negative number raised to a power that is not an integer

a USR function called before defining the starting address with DEF USR

a negative record number used on GET or PUT (file)

an improper argument used with a function or statement

a list or edit function attempted on a protected BASIC program

a line number deleted which does not exist

6 Overflow

The size of a number is too large to be represented in BASIC's number format. Integer overflow causes execution to halt. Otherwise, machine infinity with the proper sign is supplied as the result and execution continues. Integer overflow is the only kind of overflow that can be trapped.

To correct integer overflow, use smaller numbers, or change to single- or double-precision variables.

Note: If the size of a number is too small to be represented in BASIC's number format, you have an *underflow* condition. The result of an underflow is zero and execution continues without an error.

7 **Out of memory**

Your program is probably large, has many FOR loops or GOSUBs, many variables, expressions that are complicated, or complex painting. If it is available, you may set aside more stack space or memory area by using CLEAR at the beginning of your program.

8 **Undefined line number**

A line reference in a statement or command refers to a non-existent line in the program. Check the line numbers in your program and correct as needed.

9 **Subscript out of range**

You used an array element with:

a subscript that is outside the dimensions of the array

the wrong number of subscripts

Check how you are using the array variable. You may have:

used a subscript on a variable that is not an array

coded a built-in function incorrectly

10 Duplicate Definition

You attempted to define the size of the same array twice. This may happen in one of the following ways:

The same array was defined in two DIM statements.

Fix the program so each array is defined only once.

The program encountered a DIM statement for an array after the default dimension of 10 was established for the array.

Move the DIM statement to the correct position in your program.

The program detected an OPTION BASE statement after an array was dimensioned, either by a DIM statement or by default.

Move the OPTION BASE statement so that it is executed before you use any arrays.

11 Division by zero

In an expression, you attempted to divide by zero, or to raise zero to a negative power. You cannot trap this error. You do not need to correct this condition, because the program continues running.

One of the following occurs automatically:

Machine infinity with the sign of the number being divided is supplied as the result of the division.

Positive machine infinity is supplied as the result of the exponentiation.

12 Illegal direct

You tried to enter a statement in direct mode which is invalid in that mode (for example, DEF FN).

You must enter these statements as part of a program line instead of in direct mode.

13 Type mismatch

This message is caused by one of the following:

a string value where a numeric value was expected

a numeric value where a string value was expected

SWAP variables of different types such as single-precision swapped for double-precision.

14 Out of string space

BASIC allocates string space dynamically until it runs out of available memory. This message means that string variables caused BASIC to exceed the amount of available memory remaining after housecleaning.

15 String too long

You attempted to create a string more than 255 characters long.

Break the string into smaller strings.

16 String formula too complex

A string expression is either too long or too complex.

Break the expression into smaller expressions.

17 Can't continue

You attempted to use CONT to continue a program that:

halted due to an error

was modified during a break in execution

does not exist

Load the program if it is not loaded and use RUN to execute it.

18 Undefined user function

A function was called before it was defined with the DEF FN statement.

Be sure that the program executes the DEF FN statement before you use the function.

19 No RESUME

Your program branched to an active error trapping routine as a result of an error condition or an ERROR statement. The routine does not contain a RESUME statement. (The physical end of the program was detected in the error trapping routine.)

Be sure to include RESUME in the error trapping routine to continue program execution. You may also include an ON ERROR GOTO O statement to your error trapping routine so BASIC displays a message for any untrapped error.

20 RESUME without error

The program has detected a RESUME statement without having trapped an error. Enter the error trapping routine only when an error occurs or an ERROR statement is executed.

If you include a STOP or END statement before the error trapping routine it will prevent the program from "falling into" the error trapping code.

22 Missing operand

An expression contains an operator, such as OR, with no operand following it.

Be sure to include all the required operands in the expression.

23 Line buffer overflow

You attempted to enter a line that had too many characters.

Split multiple statements so they are on more than one line. You could also substitute string variables for constants where possible.

24 Device Timeout

BASIC did not receive information from an I/O device within a specified amount of time.

For communications files, this error message says that one or more of the signals you tested with OPEN "COM... was not found in the prescribed amount of time.

You should try the operation again.

25 Device Fault

Indicates a hardware error in an interface adapter.

This message may occur when you are transmitting data to a communications file. It indicates that one or more of the tested signals (as you specified on the OPEN "COM... statement) was not found within the prescribed period of time.

26 FOR without NEXT

A FOR was detected without a corresponding NEXT.

Change the program so it contains the required NEXT statement.

27 Out of Paper

The printer is either not ready or out of paper.

Insort paper (if necessary), verify that the printer is properly connected, and be sure that the power is on; then, continue the program.

29 WHILE without WEND

A WHILE statement lacks a corresponding WEND. This error occurs when a WHILE is active at the physical end of the program.

Correct the program so that each WHILE has a matching WEND.

30 WEND without WHILE

A WEND is detected before a matching WHILE was executed.

Correct the program so that you have a WHILE for each WEND.

50 FIELD overflow

A FIELD statement attempted to allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or, the end of the FIELD buffer was detected while doing sequential I/O (PRINT #, WRITE #, INPUT #) to a random file.

Verify that the OPEN statement and the FIELD statement correspond. If you are doing sequential I/O to a random file, verify that the length of the data read or written is not longer than the record length of the random file.

51 Internal error

An internal error occurred in BASIC.

Make a new copy of your BASIC diskette. Check for obvious problems with your hardware and retry the operation. If you get the error again, report the problem to your computer dealer.

52 Bad file number

One of the following has occured:

A statement used a file number of a file that was not open.

The file number was out of the range of possible file numbers specified at initialization.

The device name in the file specification was too long or invalid.

The filename was too long or invalid.

Verify that the file you wanted was opened and that the file number was entered correctly in the statement, and that you have a valid file specification.

53 File not found

A LOAD, KILL, NAME, FILES, or OPEN references a non-existent file on the diskette.

Verify that you are using the correct diskette, and that you entered the file specification correctly.

54 Bad file mode

You attempted to do one of the following:

use PUT or GET with a sequential file or a closed file

execute an OPEN with a file mode other than input, output, append, or random.

Verify that the OPEN statement was entered and executed correctly, Remember that GET and PUT require a random file. This error also occurs if you try to merge a non-ASCII file. Be sure you are merging the right file and if necessary, load the program and save it using the A option.

55 File already open

You attempted:

to open a file for sequential output or append, and the file was already opened

to use KILL on a file that was open

If you are writing to a sequential file, do not execute more than one OPEN. Before using KILL, you must close the file.

57 Device I/O Error

An error occurred on a device I/O operation (usually disk). DOS was not able to recover from the error.

When receiving communications data, this error may occur due to overrun, framing, break, or parity errors. When receiving data with 7 or less data bits, the eighth bit is turned on in the byte in error.

58 File already exists

The filename you specified in a NAME command is the same as a filename already in use on the disk.

Enter the NAME command with a different name.

61 Disk full

You have run out of your disk storage space. Files are closed when this error occurs.

If possible, erase some files on the disk, or use a new disk. Then you may retry the operation or rerun the program.

62 Input past end

This message indicates an end of file error. You attempted to execute an input statement on a null (empty) file, or after all the data in a sequential file was already input.

You can avoid this error by using the EOF function to detect the end of file.

This error will also occur if you try to read a file that was opened for output or append. If you want to read a sequential output (or append) file, you must close it and reopen it as an input file.

63 Bad record number

In a PUT or GET (file) statement, the record number is either greater than the maximum allowed (32767) or equal to zero.

In BASIC 2.0, GET and PUT have been enhanced to allow record numbers ranging from 1 to 16,777,215. This allows for large files with short record numbers.

64 Bad file name

You have used an invalid form for the filename with KILL, NAME, or FILES.

Correct the filename in error.

66 Direct statement in file

A direct statement was encountered during load or chain to an ASCII-format file. The LOAD or CHAIN is terminated due to the error.

The ASCII file should include only statements preceded by line numbers. This error may also occur if a line feed character is detected in the input stream.

67 Too many files

You attempted to create:

a new file (using SAVE or OPEN) but all directory entries on the diskette are full

an invalid file specification

Verify that the file specification is okay. If it is, use a new formatted diskette and retry the operation.

68 Device Unavailable

You attempted to open a file to a non-existent device. Either you do not have the hardware to support the specified device (such as printer adapters for a second printer), or you have disabled the device.

Verify that the device is installed correctly. You may need to enter the command:

SYSTEM

This will return you to DOS; where you can re-enter the BASIC command.

69 Communication buffer overflow

You executed a communication input statement, but the input buffer was full.

When this condition occurs, use an ON ERROR statement to retry the input. Subsequent inputs try to clear this fault unless characters continue to be received at a faster rate than the program can process. If this happens, try one of the following:

When you start BASIC, increase the size of the communications buffer using the /C: option.

Use a hand-shaking protocol with the other computer to tell it to stop sending long enough so that receive can catch up.

Specify a lower baud rate for transmit and receive.

70 Disk write protected

You attempted to write to a diskette that is write-protected.

Verify that you are using the correct diskette. If so, remove the write protection, and retry the operation.

A hardware failure may also cause this error.

71 Disk not Ready

Either the disk drive door is open or a disk is not in the drive.

Place the correct disk in the drive and close the drive door.

72 Disk media error

The controller attachment card discovered a hardware or media fault. Usually, this means that you have a faulty disk.

Copy any existing files to a new disk and attempt to format the disk. If the disk will not format, throw it away.

73 Advanced Feature

You used an Advanced BASIC Feature while you were in BASIC. Start BASICA and rerun your program.

74 Rename across disks

You renamed a file, but you specified the incorrect disk. The operation will not be performed.

75 Path/file access error

You attempted to use a path or filename for an inaccessible file during an **OPEN, NAME, MKDIR, CHDIR, or RMDIR** operation. You may have tried to open a directly or to remove the current directory; or you might have tried to open a read-only file for writing. Operation not completed.

76 Path not found

DOS cannot find the path as it is specified in an **OPEN, MKDIR, CHDIR,** or **RMDIR** operation. Operation is terminated.

—Incorrect DOS Version

The command you entered requires a different version of DOS from the one you are executing.

—Unprintable error

Your program has generated an error condition for which no error message exists. This is generally caused by an ERROR statement with an undefined error code.

Verify that your program can handle all error codes you create.

MEMO

ALPHABETIC CROSS-REFERENCE

Error Message	Number
Advanced Feature	73
Bad file mode	54
Bad file name	64
Bad file number	52
Bad record number	63
Can't continue	17
Communication buffer overflow	69
Device Fault	25
Device I/O Error	57
Device Timeout	24
Device Unavailable	68
Direct statement in file	66
Disk full	61
Disk media error	72
Disk not Ready	71
Disk write protected	70
Division by zero	11
Duplicate Definition	10
FIELD overflow	50
File already exists	58
File already open	55
File not found	53
FOR without NEXT	26
Illegal direct	12
Illegal function call	5
Incorrect DOS version	-
Input past end	62
Internal error	51
Line buffer overflow	23
Missing operand	22

Error Message	Number
Next without FOR	1
No RESUME	19
Out of data	4
Out of memory	7
Out of Paper	27
Out of string space	14
Overflow	6
Path/file access error	75
Path not found	76
Rename across disks	74
RESUME without error	20
RETURN without GOSUB	3
String formula too complex	16
String too long	15
Subscript out of range	9
Syntax error	2
Too many files	67
Type mismatch	13
Undefined line number	8
Undefined user function	18
Unprintable error	—
WEND without WHILE	30
WHILE without WEND	29

APPENDIX B
MATHEMATICAL FUNCTIONS

DERIVED FUNCTIONS

You may calculate functions that are not intrinsic to Panasonic BASIC as follows:

Function	Equivalent
Logarithm to base B	LOGB(X)=LOG(X)/LOG(B)
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(1-X*X))
Inverse cosine	ARCCOS(X)=1.570796-ATN(X/SQR(1-X*X))
Inverse secant	ARCSEC(X)=ATN(SQR(X*X-1))+(X<0)*3.141593
Inverse cosecant	ARCCSC(X)=ATN(1/SQR(X*X-1))+(X<0)*3.141593

Function	Equivalent
Inverse cotangent	ARCCOT(X)=1.57096–ATN(X)
Hyperbolic sine	SINH(X)=(EXP(X)–EXP(–X))/2
Hyperbolic cosine	COSH(X)=(EXP(X)+EXP(–X))/2
Hyperbolic tangent	TANH(X)=(EXP(X)–EXP(–X))/(EXP(X)+ EXP(–X))
Hyperbolic secant	SECH(X)=2/(EXP(X)+EXP(–X))
Hyperbolic cosecant	CSCH(X)=2/(EXP(X)–EXP(–X))
Hyperbolic cotangent	COTH(X)=(EXP(X)+EXP(–X))/(EXP(X)– EXP(–X))
Inverse hyperbolic sine	ARCSINH(X)=LOG(X+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((1+SQR(1–X*X))/X)
Inverse hyperbolic cosecant	ARCCSCH(X)=LOG((1+SGN(X)* SQR(1+X*X))/X)
Inverse hyperbolic cotangent	ARCCOTH(X)=LOG((X+1)/(X–1))/2

APPENDIX C

ASCII CHARACTER CODES

ASCII Value (decimal)	Character	Control character
000	BLANK (NULL)	NUL
001	☺	SOH
002	☻	STX
003	♥	EXT
004	♦	EOT
005	♣	ENQ
006	♠	ACK
007	•	BEL
008	◘	BS
009	○	HT
010	◙	LF
011	♂	VT
012	♀	FF
013	♪	CR
014	♫	SO
015	☼	SI
016	►	DLE
017	◄	DC1
018	↕	DC2
019	‼	DC3
020	¶	DC4
021	§	NAK
022	▬	SYN
023	↨	ETB
024	↑	CAN
025	↓	EM
026	→	SUB
027	←	ESC
028	∟	FS
029	↔	GS
030	▲	RS
031	▼	US
032	BLANK (SPACE)	
033	!	

APP. C

ASCII Value	Character
034	"
035	#
036	$
037	%
038	&
039	'
040	(
041	)
041	*
043	+
044	,
045	—
046	.
047	/
048	0
049	1
050	2
051	3
052	4
053	5
054	6
055	7
056	8
057	9
058	:
059	;
060	<
061	=
062	>
063	?
064	@
065	A
066	B
067	C
068	D
069	E
070	F

ASCII Value	Character
071	G
072	H
073	I
074	J
075	K
076	L
077	M
078	N
079	O
080	P
081	Q
082	R
083	S
084	T
085	U
086	V
087	W
088	X
089	Y
090	Z
091	[
092	\
093	]
094	^
095	_
096	‘
097	a
098	b
099	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k

ASCII Value	Character
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	¦
125	}
126	~
127	△
128	Ç
129	ü
130	é
131	â
132	ä
133	à
134	å
135	ç
136	ê
137	ë
138	è
139	ï
140	î
141	ì
142	Ä
143	Å
144	É

ASCII Value	Character
145	æ
146	Æ
147	ô
148	ö
149	ò
150	û
151	ù
152	ÿ
153	Ö
154	Ü
155	¢
156	£
157	¥
158	Pt
159	f
160	á
161	í
162	ó
163	ú
164	ñ
165	Ñ
166	a
167	o
168	¿
169	⌐
170	¬
171	½
172	¼
173	¡
174	»
175	«
176	░
177	▒
178	▓
179	│
180	┤
181	╡

APP. C

APP. C

ASCII Value	Character
182	╢
183	╖
184	╕
185	╣
186	║
187	╗
189	╨
190	╛
191	┐
192	└
193	┴
194	┬
195	├
196	─
197	┼
198	╞
199	╟
200	╚
201	╔
202	╩
203	╦
204	╠
205	═
206	╬
207	╧
208	╨
209	╤
210	╥
211	╙
212	╘
213	╒
214	╓
215	╫
216	╪
217	┘
218	┌

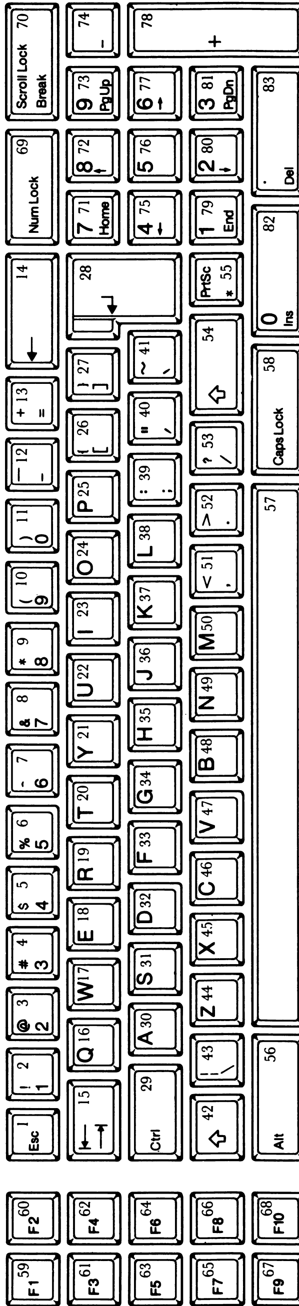
ASCII Value	Character
219	█
220	▄
221	▌
222	▐
223	▀
224	α
225	β
226	γ
227	π
228	Σ
229	σ
230	μ
231	τ
232	Φ
233	θ
234	Ω
235	δ
236	∞
237	∅
238	∈
239	∩
240	≡
241	±
242	≥
243	≤
244	⌠
245	⌡
246	÷
247	≈
248	°
249	•
250	·
251	√
252	ⁿ
253	²
254	■
255	BLANK (FF)

EXTENDED CODES

Since some keys or key combinations cannot be specified in ASCII codes, an extended code is defined for them. For instance, INKEY$ returns an extended code listed below when <Alt> <A> is pressed.

Extended Code	Meaning
3	(null character) NUL
15	(shift tab) ⇤
16-25	Alt- Q, W, E, R, T, Y, U, I, O, P
30-38	Alt- A, S, D, F, G, H, J, K, L
44-50	Alt- Z, X, C, V, B, N, M
59-68	function keys F1 through F10 (When disabled as soft keys)
71	Home
72	Cursor Up
73	Pg Up
75	Cursor Left
77	Cursor Right
79	End
80	Cursor Down
81	Pg Dn
82	Ins
83	Del
84-93	F11-F20 (Shift- F1 through F10)
94-103	F21-F30 (Ctrl- F1 through F10)
104-113	F31-F40 (Alt- F1 through F10)
114	Ctrl-PrtSc
115	Ctrl-Cursor Left (Previous Word)
116	Ctrl-Cursor Right (Next Word)
117	Ctrl-End
118	Ctrl-Pg Dn
119	Ctrl-Home
120-131	Alt- 1,2,3,4,5,6,7,8,9,0.-,=
132	Ctrl-Pg Up

Key Position



Nomenclature is on both the top and front face of the keybutton. The number on the upper right of the keybutton is key position number. Reffer to page C-7, the relationship between keyposition and scan code is shown.

Key Position and Scan Codes

Key Position	Scan code (&H)	Key Position Number	Scan Code (&H)
1	01	43	2B
2	02	44	2C
3	03	45	2D
4	04	46	2E
5	05	47	2F
6	06	48	30
7	07	49	31
8	08	50	32
9	09	51	33
10	0A	52	34
11	0B	53	35
12	0C	54	36
13	0D	55	37
14	0E	56	38
15	0F	57	39
16	10	58	3A
17	11	59	3B
18	12	60	3C
19	13	61	3D
20	14	62	3E
21	15	63	3F
22	16	64	40
23	17	65	41
24	18	66	42
25	19	67	43
26	1A	68	44
27	1B	69	45
28	1C	70	46
29	1D	71	47
30	1E	72	48
31	1F	73	49
32	20	74	4A
33	21	75	4B
34	22	76	4C
35	23	77	4D
36	24	78	4E
37	25	79	4F
38	26	80	50
39	27	81	51
40	28	82	52
41	29	83	53
42	2A		

MEMO

APP. C

APPENDIX D
ACCESSING MACHINE LANGUAGE SUBROUTINES

INTRODUCTION D-2

ALLOCATING MEMORY SPACE FOR SUBROUTINES D-3

PLACING A SUBROUTINE IN MEMORY D-5

Using the POKE Command D-5
Loading a Subroutine File D-7

ACCESSING A SUBROUTINE DURING EXECUTION D-9

The CALL Statement D-10
USR Functions D-11

APP. D

INTRODUCTION

During execution, BASIC programs may access machine language subroutines that have been stored in memory. This appendix describes how subroutine code is both placed in memory and subsequently used by programs.

APP. D

ALLOCATING MEMORY SPACE FOR SUBROUTINES

DOS requires 25K bytes and BASIC a further 52K bytes. In additon, BASIC normally uses 64K bytes of memory to contain your program and the data it requires, as well as the interpreter's workarea and the BASIC stack. You can store subroutines either outside this area, or you can reserve space within BASIC's workarea if you have determined that there will be enough.

If you want to store subroutines beyond the 141K-byte area needed by DOS and BASIC, you will need at least a 256K-byte system in order to have room for them. To store subroutines outside the DOS/BASIC workarea, you need only specify the high memory address where the first byte of subroutine code will be placed. This is done with the DEF SEG statement. The following example statement specifies a hex location in the last 4K bytes of a 256K-byte system, where subsequent loading of subroutine code will begin.

```
100 DEF SEG=&H3F00
```

You can also set aside space for subroutines at the upper end of BASIC's 64K-byte workarea, if you determine that your program and data, plus the BASIC functions, will not require all the available 64K bytes. This can be done either with the CLEAR command, or by using the M: option of the BASIC statement.

The following examples show two ways in which the upper 4K bytes of BASIC's workarea can be reserved for subroutines, if you use Sr. Partner with more than 256K bytes. You can use the CLEAR command:

```
100 CLEAR, &HF000
```

Or you can use the BASIC command:

```
BASIC/M:&HF000
```

Both commands set BASIC's workarea size to hex F000, or 60K, bytes.

If you use Sr. Partner with 128K bytes, only 50K bytes BASIC's workarea are reserved. Space for subroutines must be in this workarea. The following example shows how to reserve the last 4K bytes for subroutines.

```
    CLEAR, FRE(0)-4096
100 DEF SEG=&H1F00
```

PLACING A SUBROUTINE IN MEMORY

Two common ways of placing a subroutine in memory are:

using the POKE command in your program

loading a disk file containing the subroutine code

Using the POKE Command

This method of entering code into memory is used most often for short subroutines. The subroutine and its loading actually become part of the BASIC program.

First, code your subroutine in machine language. Then, write DATA statements that specify the hexadecimal equivalent of each byte of your code.

After defining the area where storage of the subroutine is to begin (unless you have done so with the BASIC statement M: option), have your program perform a loop in which each data byte is read, then use the POKE command to place it into the subroutine space.

When the loop is complete, the subroutine is loaded. If you use a USR function to access it, then include a DEF USR statement to define the address where the subroutine begins. If you are going to CALL the subroutine, then set the SUB variable to the entry address.

The following example program shows how process may be programmed.

The letters A to Z represent integer variable values:

```
100 DEFINT A-Z
```

Start loading the subroutine at hex location 1F00, if you use 128K-byte system:

```
105 DEF SEG=&H1700
```

APP. D

Perform a loop 22 times to load 22 bytes of code:

```
110 FOR X=0 TO 21
```

Read a byte of data and store it in the address X bytes from the start of the subroutine area:

```
115 READ Y
120 POKE X,Y
```

Continue the loop until it is finished:

```
125 NEXT
```

Set the entry address for the subroutine:

```
130 SUB=0
```

Assign values to three variables:

```
135 V1=2:  V2=5:  V3=0
```

Start executing the subroutine using V1, V2, and V3 as arguments:

```
140 CALL SUB (V1, V2, V3)
```

When the subroutine has executed, print V3, which is the returned result:

```
145 PRINT V3
```

Stop executing:

```
150 END
```

This is the subroutine code to be read and loaded with the loop:

```
155 DATA &H55,&H8B,&HEC,&H8B,&H76,&H0A
160 DATA &H8B,&H04,&H8B,&H76,&H08
165 DATA &H03,&H04,&H8B,&H7E,&H06
170 DATA &H89,&H05,&H5D,&HCA,&H06,&H00
RUN
7
OK
```

Loading a Subroutine File

Converting a routine longer than 22 bytes of code the hexadecimal, as well as entering the hex values, would be a tedious, time-consuming process. Fortunately, this can be done for you by the DOS linker, which, given machine language, will produce from it an executable memory image (that is, hex) file. This file may then be saved, and loaded at any time from a BASIC program and accessed either by a USR function or a CALL statement.

The process of loading the linked file is more complicated than loading a POKE loop. However, writing a machine language program, while not simple, is much easier than converting hundreds of bytes to hex and the entering the DATA statements to feed the POKE loop.

The following procedure can be used to load a subroutine that has been linked so that it is loaded at the high end of memory and subsequently access it from a BASIC program.

1. Access BASIC in DEBUG mode by entering:

   ```
   DEBUG BASICA.COM/M:10000
   ```

   This causes DEBUG to precede BASIC in memory, so that DEBUG will not be over-written. In DEBUG mode, subroutine breakpoints, if used, will cause a return to DEBUG, as will a SYSTEM command entered from BASIC.

2. Ascertain where BASIC was placed in memory by entering the DEBUG R command, which displays the contents of the registers CS, IP, SS, SP, DS, and ES. Record the value of each register for later use.

3. Enter the following commands to name and then load the linker file:

   ```
   N SUBNAME.EXE
   L
   ```

   The filename is your choice; the extension .EXE causes the file to be loaded at the high end of memory.

4. Display the registers again, to determine where the subroutine was placed. Record the contents of the CS and IP registers.

5. Use the R command to reset the registers to the values you recorded in Step 2 (after BASICA.COM was initially loaded).

6. If you want to set breakpoints in the subroutine, branch to the BASIC entry point with the G command.

7. Respond to BASIC's prompt (Ok) by loading your BASIC program. Edit the DEF SEG statement so that it references the CS register value recorded in Step 4. Change either the DEF USR offset or the CALL variable to the IP register value. Now the statements specify the correct address of the subroutine.

8. Save the subroutine with a BSAVE command. Specify the offset as the CS and IP values that defined the subroutine's entry point. For the length, use the code length given on the assembler listing or LINK map.

9. Insert a BLOAD statement into the BASIC program, following the DEF SEG that you edited to specify the CS register value. A self-relocatable subroutine can be loaded into a different storage location at execution time using the BLOAD command. If you do this, be certain to edit the DEF SEG accordingly.

10. Use the SAVE command to save your modified BASIC program.

ACCESSING A SUBROUTINE DURING EXECUTION

You may access a subroutine from a program in one of two ways, either with a CALL statement or a USR function. Whichever method you use, the following conditions apply:

Upon entering the subroutine, the DS, ES and SS segment registers all contain the same value, the address of BASIC's data area.

The code segment (CS) register contains either the same value as the other registers, or an address specified by the last DEF SEG.

If a string argument is passed from the program, the value that is received by the subroutine is the address of the string descriptor, which is three bytes containing the following information:

* the length of the string (Byte 0)

* the offset from the beginning of BASIC's data area where the string is located (Byte 1, 2)

  The subroutine must not change the content of the string descriptor. The content of the string may be altered, but not its length or location.

When returning from the subroutine:

* Before the return, restore the contents of the segment registers and, if necessary, the stack pointer (SP). (At entry, the stack pointer indicates a 16-byte stack available to the subroutine. If it needs more, the subroutine must set up its own stack segment and pointer, recording the current stack location and restoring it before the return.)

* Interrupts disabled by the subroutine must be re-enabled before the return.

* Causes the return to BASIC with an inter-segment RET instruction.

The CALL Statement

A subroutine may be accessed with the call statement, which allows multiple arguments to be passed. A CALL statement is formatted as follows:

```
CALL NVAR [(variable1, variable2,...)]
```

NVAR is a numeric variable whose value is the offset from the point in memory specified by DEF SEG where the subroutine is stored. (Remember the POKE loop program? The SUB variable was the NVAR there.)

The variables are the arguments passed to the subroutine. The square brackets mean that their number is optional, meaning: the number of arguments is determined by the requirements of a particular subroutine.

These variables, which must be separated by commas, may not be constants. They must be either numeric or string variables.

When a CALL is executed, the following sequence occurs:

1. The location (an offset into BASIC's DS) of each variable in the variable list is pushed onto the stack. In the case of a string variable, this address is that of the string descriptor.

2. The return address in the CS register, plus the offset, are also pushed onto the stack.

3. Control is passed to the subroutine, using the address specified by the last DEF SEG, plus the offset indicated by NVAR.

The subroutine must know the number of arguments that were passed. Parameters are referenced by adding as offset to BP after the subroutine moves the stack pointer into BP. For this purpose, the first subroutine instructions must be:

```
PUSH BP          ;SAVE BP
MOV BP, SP       ;MOVE SP TO BP
```

An argument's offset into the stack is computed with the following formula:

```
Offset from BP=2*(n-m)+6
```

In this statement, n is the total number of arguments, and m is the position of an argument in the variable list of the CALL statement, a number that may range from 1 to n.

USR Functions

A function is an operation performed on no more than one argument per iteration. A USR (user-defined) function is coded as a subroutine and accessed by a BASIC program similarly to a subroutine accessed by using the CALL statement. A USR function call is formatted as follows:

```
USR[n](arg)
```

In this statement, n is a number from zero through nine that identifies the desired USR function. It is the same number specified by the corresponding DEF USR statement, which gives the offset from the last DEF SEG where the subroutine is located. USR0 is the default if n is not included.

The argument must be either a string variable name or a numeric expression. When calling a function for which arguments are not needed, a dummy argument must be supplied.

The following code sequence shows how a USR function can be accessed during execution of a program:

```
10 DEF SEG=&H1800
15 BLOAD "SUB.EXE",0
20 DEF USR0=0
25 X=2
30 Y=USR0(X)
35 PRINT Y
```

The subroutine is loaded starting at hex location 18000, and will be referenced as USR0. The function is called to compute a value for Y, with the variable used in the operation to be X.

When a USR function is accessed, a number specifying the type of argument is placed in register AL. These values, and the argument types they indicate, are as follows:

AL Value	Argument Type
2	Integer
3	String
4	Single-precision Number
8	Double-precision Number

When a string argument is passed, the DX register contains the address of the string descriptor.

APP. D

The value of a numeric argument is placed in the eight-byte FAC (Floating Point Accumulator), and the BX register is set to contain the offset into BASIC's data area where the fifth byte of the FAC is located.

The following explanatory examples are based on the FAC being located in (hex) bytes 5FC through 603, with BX pointing to Byte 600.

An *integer argument* will be placed in Bytes 600 and 601.

For *a single-precision argument*:

Bytes 600 and 601 will contain, respectively, the lowest and middle eight bits of the mantissa.

Byte 602 will contain the highest seven bits of the mantissa. The leading bit is suppressed, and the last bit contains, instead, the number's sign (zero for plus and 1 for minus).

Byte 603 will contain the exponent minus 128, with the binary point to the left of the mantissa's most significant bit.

A *double-precision argument* is placed so the Bytes 600 through 603 are the same as for a single-precision number. Additionally, Bytes 5FC through 5FF are used to contain another four bytes of mantissa (the lowest eight bits in 5FC).

A USR function will usually return a result of the same type as the argument. However, a single- or double-precision argument can be forced to its integer equivalent with the FRCINT routine, before the function calculation is done. If an integer value must be returned, place the computation result in the BX register, then call the MAKINT routine prior to the return, which places the integer value in the FAC to passed back to BASIC.

The following example shows the methods for accessing FRCINT and MAKINT:

```
10 DEF SEG=&H1F00
20 BLOAD "SUB.EXE",0
30 DEF USR0=0
40 A#=100
50 B=USR0(A#)
60 PRINT B
```

APP. D

The following Macro-Assembler language subroutine has been loaded at location 1F00:0. It doubles the argument passed and returns an integer result.

```
CDSEG     SEGMENT
          ASSUME    CS:CDSEG

FRCOFF    EQU       0F0H
MAKOFF    EQU       0F04H

FRCINT    LABEL     DWORD
          DW        FRCOFF
FRCSEG    DW        ?

MAKINT    LABEL     DWORD
          DW        MAKINT
MAKSEG    DW        ?

USRPRG    PROC      FAR
          POP       CX
          POP       DI             ; Recover BASIC's CS
          PUSH      DI
          PUSH      CX
          MOV       FRCSEG, DI     ; Set segment for CALL
          MOV       MAKSEG, DI
          CALL      FRCINT
          ADD       BX,BX          ; Force ARG in FAC into[BX]
          CALL      MAKINT
          RET
USRPRG    ENDP
CDSEG     ENDS                     ; Put INT Result in BX into
                                     FAC
```

When FRCINT or MAKINT is called, and when it terminates with a return, ES, DS and SS must have the same value which they had when the USPRG was entered. They point to the BASIC's data segment.

APPENDIX E
CONVERTING A PROGRAM TO PANASONIC BASIC

PANASONIC AND OTHER BASICS E-2

- File I/O Processing E-2
- Graphics Capability E-2
- IF...THEN Statements E-2
- Line Feeds E-3
- Logical Operations E-4
- MAT Functions E-6
- Multiple Assignment Statements E-6
- Multiple Statements on a Line E-6
- PEEK and POKE Statements E-6
- Results of Relational Comparisons E-6
- Remarks E-7
- Rounding Numbers E-7
- Ringing the Bell E-7
- String Handling E-7
- Delimiting Keywords E-8

APP. E

PANASONIC AND OTHER BASICS

The version of BASIC recognized by your Panasonic is very similar to the BASICs used to program many other small computers. A program written in a different BASIC can usually be run on the Panasonic with only minor changes. This appendix describes some of the areas where variations of purpose or functioning may make such adjustments necessary.

File I/O Processing

When a Panasonic disk file is opened for input or output purposes, a specific file number is associated with it. This number is then referenced by I/O statements that follow. Some other BASICs implement file I/O processes differently.

Panasonic BASIC also routinely blocks random access file records whenever appropriate, so that as many records as possible may be fitted into a sector. This may or may not be the case with other BASICs.

APP. E

Graphics Capability

The graphic possibilities of different BASICs vary widely. A figure specified by a program written in a different BASIC may be either more or less that Panasonic BASIC can draw.

IF...THEN Statements

Panasonic BASIC's IF statement can include an optional ELSE clause, which is performed when the tested expression is false. This is not the case with some other BASICs. For instance, a decision process may be expressed in another BASIC as:

```
100 IF X=Y THEN 300
200 PRINT "UNEQUAL" : GOTO 400
300 PRINT "SAME NUMBER"
400 GOTO 50
```

This sequence of statements will function in the same way on the Panasonic, but could also be written more concisely in Panasonic BASIC as:

```
100 IF X=Y THEN PRINT "SAME NUMBER" ELSE PRINT "UNEQUAL"
200 GOTO 50
```

The Panasonic IF statement also permits THEN and ELSE clauses containing multiple statements, which can bring about a different result when a sequence written in a different BASIC is executed. For example:

```
100 IF X=Y THEN GOTO 500 : PRINT "UNEQUAL"
```

In some other BASICs, if a test fails, the program continues with the next *statement*. So, in the above example, if X is not equal to Y, the PRINT statement will be executed an "UNEQUAL" will be printed.

When a test fails in Panasonic BASIC, the program continues with the next *line*. And, since Panasonic BASIC considers both the GOTO and PRINT statements to be included in the THEN clause, if the test fails, the program will continue on the succeeding line, and the PRINT statement will never be executed. The example statements (Line 100) may be rewritten to function correctly on the Panasonic as:

```
100 IF X=Y THEN 500 ELSE PRINT "UNEQUAL"
200 GOTO 50
```

Line Feeds

Some other BASICs place a "line feed character" in the input text when a line feed is specified. Panasonic BASIC inserts blank characters from the place the line feed was indicated to the end of the display line. If you attempt to load a program that contains line feed characters, Panasonic BASIC will issue a "Direct Statement in File" error message.

Logical Operations

In some BASICs, a logical operation determines whether the operands are either zero (indicating a "false" expression) or non-zero (indicating "true"). The following statements, written in another BASIC, show how this process works:

```
100 X=9 : Y=4
200 IF X AND Y THEN PRINT "BOTH ARE TRUE"
```

The other BASIC analyzes the logical expression "9 AND 4" as follows:

1. 9 is a non-zero value, so it is "true".

2. 4 is also not zero, so it is also "true".

3. Since both values are non-zero, and "true", it follows, via AND logic, that the logical expression is "true".

In the other BASIC, the computer prints "BOTH ARE TRUE".

APP. E

In Panasonic BASIC, logical operands are numeric values. Though −1 and zero are used to specify the "true" or "false" results of relational comparisons, logical operations can also be applied to any integer values. Thus, when Panasonic BASIC analyzes any AND operation, it converts the operands to their binary equivalents and analyzes these values according to the rules of AND logic. These rules state that the result of an AND operation is "true" only if both expressions are true. In binary terms then, only two ones can result in a one.

So when Panasonic BASIC sees the logical expression "9 AND 4", it analyzes it this way:

1. The binary equivalent of 9 is:

   1001

2. The binary equivalent of 4 is:

   0100

3. Combining each vertical pair of digits according to AND logic gives the result:

```
      9=1001
      4=0100
RESULT=0000
```

That is, a one and a zero result in a zero, and the result of two zeros is a zero.

Thus, Panasonic BASIC will conclude by the "false" (zero) result that the AND expression is not true, and will print nothing.

The results desired in the example could be achieved with Panasonic BASIC by a more direct statement of what you want to know. You could ask the question, "Are both these values non-zero?". This could be done in the following way:

```
100 X=9:Y=4
200 IF (X<>0) AND (Y<>0) THEN PRINT "BOTH NON-ZERO"
```

Here, Panasonic BASIC ANDs the results of the relational operations in parentheses. Both results would be "ture", since 9 is indeed unequal to zero, and it is also true that 4 is unequal to zero. So Panasonic BASIC would analyze the expression "−1 AND −1" this way:

1. The binary equivalent of −1 is 1111111111111111

2. Combining this with another −1 according to AND logic gives the result:

```
     -1= 1111111111111111
     -1= 1111111111111111
RESULT= 1111111111111111
```

Panasonic BASIC concludes that the AND expression is "true" and prints the message.

MAT Functions

A program that uses the MAT functions included in some BASICs must have these portions rewritten with the FOR...NEXT loops used in Panasonic BASIC.

Multiple Assignment Statements

Some BASICs permit a single value to be assigned to multiple variables in one statement, as follows:

```
100 LET X=Y=5
```

This statement, in these other BASICs, assigns the value of 5 to both X and Y. In Panasonic BASIC, this must be done with two assignments. The following statement is an example of a simple way to do this:

```
100 X=5 : Y=5
```

APP. E

Multiple Statements on a Line

In some BASICs, the backslash (\) is used to separate multiple statements on a single program line. Panasonic BASIC requires that the colon (:) be used to do this.

PEEK and POKE Statements

PEEK and POKE statements are used for varying purposes in different BASICs. Be certain that a PEEK or POKE statement in a program written in another BASIC is used for the same function that you want Panasonic BASIC to perform.

Results of Relational Comparisons

In Panasonic BASIC, the results of a relational comparison are indicated by −1 if the expression is true, and by zero if the expression is false. Some other BASICs use a positive 1 to specify a "true" result. This can affect the result of an arithmetic computation that includes the relational expression.

Remarks

Some BASICs use the exclamation point (!) to indicate remarks added to the end of a program line. In Panasonic BASIC, this is done with a single quote (').

Rounding Numbers

For accuracy, Panasonic BASIC rounds single- and double-precision numbers when an integer value is required. Many other BASICs truncate in such situations. This can affect not only assignments (NUM%=2.5 results in NUM%=3 in Panasonic BASIC), but also the results returned by statement evaluations and a number of functions. Rounding may also cause Panasonic BASIC to select a different element from an array — one that may be out of range, and will certainly be incorrect.

Ringing the Bell

Some BASICs use PRINT CHR$(7) to make a bell ring. Though not required, in Panasonic BASIC this may be replaced with a BEEP for a similar effect.

String Handling

Character strings in Panasonic BASIC are variable length, which means that their length is determined by the length of the strings assigned to them. Any statements that define string lengths, which may be used in other BASICs, must be modified. The following statement might be used in some BASICs to define an array made up of J string elements, each of I length:

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

However, in Panasonic BASIC, only the following would be used:

```
DIM B$(J)
```

This would dimension an array of J string elements, each of which would have an initial length = 0. The length of each element would be established when a value is assigned to it.

Some BASICs use a comma (,) or an ampersand (&) to specify concatenation of strings. This operation is specified in Panasonic BASIC by the plus sign (+).

Panasonic BASIC uses the LEFT$, MID$, and RIGHT$ functions to extract substrings from existing strings. LEFT$ accesses a substring that begins with the leftmost position of the existing string, MID$ is used to access a string starting at a point within the existing string, and RIGHT$ extracts a string that ends at the rightmost position of the existing string. LEFT$ and RIGHT$ are performed on operands that specify the name of the existing string, and the number of characters to be extracted. MID$ also requires that the position of the first character of the substring be specified.

This is done differently in some other BASICs. For example, a form such as STR$(I) might be used to access the Ith character of STR$. This should be changed to MID$ (STR$, I, 1), which references a single-character substring at the Ith position of STR$. Or, to extract a substring of STR$ that starts at position L and ends at position M, the form STR$ (L,M) might be used in other BASICs. You should change this form to:

APP. E

```
MID$(STR$, L, M-L+1)
```

Delimiting Keywords

In some BASICs, statements may be written without separating the keywords, as follows:

```
100 FOR I=NTOX
```

In Panasonic BASIC, all keywords must be delimited by spaces:

```
100 FOR I=N TO X
```

APPENDIX F EXECUTING APPLICATION PROGRAMS

When executing an application program for the IBM Personal Computer, it may not be run as described in the application manual. In such cases, an error message will be displayed. This Appendix contains possible error messages and the procedure to recover from the error. Check the error message and execute the appropriate procedure described below.

1. "Insufficient disk space" was displayed when you executed "COPY BASICA.COM.B:"

   Copy BASICA.EXE to the disk instead of BASICA.COM.

2. "Insufficient disk space" was displayed when you executed the batch file for an initialization such as UPDATE, SETUP, INSTALL etc. (It depends on application program).

   This error message is displayed because the batch file for an initialization tried to COPY BASICA.COM. to the disk without enough space for it. After the DOS prompt appears, type COPY BASICA.EXE B: and execute the bath file (UPDATE, SETUP, INSTALL etc.) again. "Insufficient disk error" will be displayed again, but ignore it.

3. "Insert Panasonic System Disk" was displayed when you executed an application progam.

   This message is displayed when you execute BASIC or BASICA on your application program disk that does not contain BASICA.COM. It may contain BASIC.COM. or BASICA.EXE. Remove the application program disk and insert your back up Panasonic System Disk. Press any key to load BASICA.COM. When "Replace the disk" appears, reinsert your application program disk. Then the application program will be executed.

   > **Note:** When you copy BASIC.COM on an application program disk, also copy BASICA.COM if that disk has sufficent space. Then the procedure of removing and inserting disks can be omitted.

4. An application program was not executed though the memory (RAM) size was same as that designated by the application manual.

   This is because of our BASIC's specific structure. Expand memory size by using the Panasonic RAM Board.

APP. F

APPENDIX G COMMUNICATION I/O PROCEDURES

INTERFACE WITH OTHER DEVICES G-2

Opening and Using a COM File G-2
Variations in Accessing COM Files G-2

APP. G

INTERFACE WITH OTHER DEVICES

BASIC can initiate and monitor communication between your Panasonic computer and other devices, a process called "asynchronous communication", which means that characters are transmitted from one device to another bit by bit. The details of this operation, and there are many, are discussed fully in Chapter 7 (see the OPEN "COM... statement).

The focus of this appendix is the programming aspects of the I/O process, as they apply to reading data from other computers, or sending output to, for instance, a printer.

Opening and Using a COM File

Communication input and output are either similar or identical both in theory and in many aspects of practice to disk file I/O (as discussed in Chapter 5).

The OPEN "COM... statement is used to allocate an I/O buffer area in the same way as OPEN accesses disk files. Since the internal serial interface or the communications adapter is opened as a file, the same I/O commands are used for communications I/O as for disk file I/O procedures:

```
INPUT #         PRINT#
LINE INPUT #    PRINT # USING
INPUT$#         WRITE#
```

APP. G

Variations in Accessing COM Files

GET and PUT

When using the GET and PUT statements, instead of the record number to be read/written, specify the number of bytes to be transferred from/to the buffer. (This number may not exceed that specified by the LEN option of the OPEN "COM... statement.)

Suspending Transmission

Sometimes characters cannot be processed as quickly as received. Transmission from another computer can be suspended while BASIC processes the buffer. If compatible with the other device, you can send XOFF (CHR$(19)) to stop transmission, and XON (CHR$(17)) to resume when the buffer is clear. XOFF tells the other computer to stop sending, and XON tells it to start again.

COM Files Functions

These COM file I/O functions can be used to detect an impending "overrun" of the input buffer capacity:

LOC returns the number of characters not yet read from the buffer. If this number is more that 255, LOC will return 255.

LOF will return the amount of free space in the buffer. This will equal n-LOC, where n is the buffer size. If not specified by the /C: option of the BASIC command, n defaults to 256.

EOF will return −1 (true) if the buffer is empty, and zero (false) if any characters remain to be read.

INPUT$ Function

When reading COM files, the INPUTS$ function will usually function faster and more efficiently than INPUT # and LINE INPUT #. The latter stops input when a carriage return is detected; INPUT # also stops if a comma is seen.

INPUT$ assigns all characters read to a string. Since a string may not contain over 255 characters, no more than that will be read at one time, preventing any overflow condition. INPUT$ can be used as follows:

```
A$=INPUT$(LOC(1), #1)
```

This statement causes the characters in the buffer to be read into A$ for processing, at a maximum rate of 255 characters per execution.

MEMO

APPENDIX H
EXAMPLE PROGRAMS

EXAMPLE PROGRAM #1 .... **H-2**

EXAMPLE PROGRAM #2 .... **H-4**

EXAMPLE PROGRAM #1

```
10 'Demonstration program of graphics commands
20 'Each demo waits 2 seconds between screen
30 '
40 SCREEN 1:KEY OFF:COLOR 4,1
50 GOTO 110
60 'The following subroutine clears the screen and centers LABEL$ on line 25
70 FOR I=1 TO 3000:NEXT I 'wait a while
80 CLS
90 LOCATE 25,<40-LEN(LABEL$))/2:PRINT LABEL$;
100 RETURN
110 '
120 'Beginning of program
130 LABEL$="Drawing points and lines":GOSUB 70
140 'Draw a rectangle of dots
150 FOR X=20 TO 310 STEP 10:PSET(X,80):PSET(X,120):NEXT X
160 FOR Y=80 TO 120 STEP 5:PSET(20,Y):PSET(310,Y):NEXT Y
170 'Draw a skewed rectangle
180 LINE(50,50)-(260,60):LINE-(250,170):LINE-(40,160):LINE-(50,50)
190 'Draw a line, then erase some points on it
200 LINE(20,20)-(320,20)
210 FOR X=25 TO 315 STEP 10:PRESET(X,20):NEXT X
220 LABEL$="Circles and an arc":GOSUB 70
230 'Draw a few circles
240 FOR X=40 TO 240 STEP 100:CIRCLE(X,40),20:NEXT X
250 'Draw an arc
260 CIRCLE(220,100),20,,0,.75*3.14159
270 'Draw a few ellipses
280 FOR X=40 TO 240 STEP 100:CIRCLE(X,140),20,,,,160/X:NEXT X
290 LABEL$="Boxes":GOSUB 70
300 'Draw a box (not filled)
310 LINE(5,5)-(310,80),,B
320 'Draw a box (filled)
330 LINE(5,90)-(310,165),,BF
340 LABEL$="Painting":GOSUB 70
350 FOR I=0 TO 200 STEP 100
360 LINE(5+I,175)-(100+I,175):LINE-(52+I,5):LINE-(5+I,175)
370 NEXT I
380 'Simple painting
```

```
390 PAINT(50,100)
400 'Painting with a pattern
410 PAINT(150,100),CHR$(&HC4)+CHR$(&H7A)
420 'Painting with another pattern
430 PAINT (250,100),CHR$(&H11)+CHR$(&HCC)+CHR$(&H33)
440 LABEL$="The DRAW Command":GOSUB 70
450 RECT1$="d20r40u20l40":RECT2$="f20e40h20g40"
460 ARROW$="r20d5e5h5d5":ARMV$="bm+6,+20"
470 DRAW "bm5,30"+RECT1$
480 DRAW "bm5,100"+RECT2$
490 DRAW "bm151,150"+ARROW$+"a1"+ARMV$+ARROW$
500 DRAW "a2"+ARMV$+ARROW$+"a3"+ARMV$+ARROW$
510 FOR I=1 TO 3000:NEXT I 'wait a while
520 LOCATE 1,1
```

EXAMPLE PROGRAM #2

```
10 'Demonstration program of random files
20 'A name and address file (up to 50 records) is used
30 'Unused records are marked with ASC(255) as the first letter in field 1
40 CLS:WIDTH 80
50 OPEN "randtest.dat" as #1:LEN=100
60 FIELD #1,25 AS CUST$,25 AS ADDR1$,25 AS ADDR2$,15 AS CITY$
,5 AS STATE$,5 AS ZIP$
70 PRINT:PRINT "Select an option:"
80 PRINT "1—add a record"
90 PRINT "2—change a record"
100 PRINT "3—look at a record"
110 PRINT "4—initialize the data file"
120 PRINT "5—list the active records"
130 PRINT "6—leave this program":PRINT
140 INPUT "Your choice:",CHOICE%
150 IF (CHOICE%<1) OR (CHOICE%>6) THEN GOTO 70
160 ON CHOICE% GOSUB 180,340,510,590,690,780
170 GOTO 70
180 'Subroutine to add a record
190 PRINT:INPUT "Which record number do you want to add (1-50):",
REC.NO%
200 IF (REC.NO%<1) OR (REC.NO%>50) THEN GOTO 180
210 GET #1,REC.NO%
220 IF ASC(CUST$)<>255 THEN PRINT "That record already exists.":
RETURN
230 INPUT "Name:",IN.CUST$
240 INPUT "Address line 1:",IN.ADDR1$
250 INPUT "Address line 2:",IN.ADDR2$
260 INPUT "City:",IN.CITY$
270 INPUT "State:",IN.STATE$
280 INPUT "Zip code:",IN.ZIP:'Notice that this is a real number
290 LSET CUST$=IN.CUST$:LSET ADDR1$=IN.ADDR1$:
LSET ADDR2$=IN.ADDR2$
300 LSET CITY$=IN.CITY$:LSET STATE$=IN.STATE$
310 LSET ZIP$=MKS$(IN.ZIP)
320 PUT #1,REC.NO%
330 RETURN
```

```
340 'Subroutine to change a record
350 PRINT:INPUT "Which record number do you want to change (1-50):"
,REC.NO%
360 IF (REC.NO%<1) OR (REC.NO%>50) THEN GOTO 340
370 GET #1,REC.NO%
380 IF ASC(CUST$)=255 THEN PRINT "That record does not have values.
":RETURN
390 PRINT "Press ENTER to leave a value the same."
400 PRINT "Name:"+CUST$+" ";:INPUT "",IN.CUST$:IF IN.CUST$=""
THEN IN.CUST$=CUST$
410 PRINT "Address line 1:"+ADDR1$+" ";:INPUT "",IN.ADDR1$:IF
IN.ADDR1$="" THEN IN.ADDR1$=ADDR1$
420 PRINT "Address line 2:"+ADDR2$+" ";:INPUT "",IN.ADDR2$:IF
IN.ADDR2$="" THEN IN.ADDR2$=ADDR2$
430 PRINT "City:"+CITY$+" ";:INPUT "",IN.CITY$:IF IN.CITY$="" THEN
IN.CITY$=CITY$
440 PRINT "State:"+STATE$+" ";:INPUT "",IN.STATE$:IF IN.STATE$=""
THEN IN.STATE$=STATE$
450 PRINT "Zip code:"+STR$(CVS(ZIP$))+" ";:INPUT "",IN.ZIP:IF
IN.ZIP=0 THEN IN.ZIP=CVS(ZIP$):'Notice that this is a real number
460 LSET CUST$=IN.CUST$:LSET ADDR1$=IN.ADDR1$:
LSET ADDR2$=IN.ADDR2$
470 LSET CITY$=IN.CITY$:LSET STATE$=IN.STATE$
480 LSET ZIP$=MKS$(IN.ZIP)
490 PUT #1,REC.NO%
500 RETURN
510 'Subroutine to display a record
520 PRINT:INPUT "Which record number do you want to display (1-50):",
REC.NO%
530 IF (REC.NO%<1) OR (REC.NO%>50) THEN GOTO 510
540 GET #1,REC.NO%
550 IF ASC(CUST$)=255 THEN PRINT "That record does not have values."
:RETURN
560 PRINT CUST$:PRINT ADDR1$:PRINT ADDR2$
570 PRINT CITY$;", ";STATE$;" ";CVS(ZIP$)
580 RETURN
590 'Subroutine to initialize the data file
600 PRINT "Are you ***sure*** you want to initialize the file?"
610 PRINT "If there are any records in the file, they will be lost."
620 INPUT "Proceed (Y or N)?",INITYN$
```

```
630 IF (INITYN$<>"Y") AND (INITYN$<>"y") THEN RETURN
640 LSET CUST$=CHR$(255):'This is the flag for unused records
650 FOR I%=1 TO 50
660 PUT #1,I%
670 NEXT I%
680 RETURN
690 'Subroutine to display active records
700 PRINT "Active records marked with an asterisk"
710 PRINT "         1         2         3         4         5"
720 PRINT "12345678901234567890123456789012345678901234567890"
730 FOR I%=1 TO 50
740 GET #1,I%
750 IF (ASC(CUST$)=255) THEN PRINT " "; ELSE PRINT "*";
760 NEXT I%
770 RETURN
780 'Leave the program
790 CLOSE
800 END
```

APP. H

APPENDIX I

INDEX

A

ABS Function 7-4
absolute form for
 specifying coordinates 6-27
accuracy 6-5
adding program lines 4-9
addition 6-15
alphabetic characters 2-20
Alt 3-5
Alt-key words 3-5
AND 6-20
arctangent 7-6
arithmetic operators 6-15
arrays 6-12
ASC Function 7-5
ASCII codes C-1
aspect ratio 7-24
ATN Function 7-6
AUTO Command 7-7
automatic line numbers 7-7

B

background 7-32, 7-34
Backspace 3-4
BASICA 2-2
BASIC, accessing 2-7
BASIC's Data Segment D-10
BEEP Statement 7-9
BLOAD Command 7-10
BSAVE Command 7-12

C

CALL Statement 7-14
cancelling a line 4-8
Caps Lock 3-4
CDBL Function 7-15
CHAIN Statement 7-16
changing BASIC program
 4-10
changing line numbers 7-216
character set 2-20
CHDIR 7- 19
CHR$ Function 7-21
CINT Function 7-22
CIRCLE Statement 7-23
CLEAR Command 7-26
clear screen 7-31
clearing memory 7-26
clock 6-28
CLOSE Statement 7-29
CLS Statement 7-31
COLOR
 in graphics mode 7-34
 in text mode 7-32
COM(n) Statement 7-36
Commands 2-18
comments 7-3
COMMON Statement 7-37
communications
 buffer size 2-9
 trapping 7- 36
comparisons
 numeric 6-17
 string 6-17
complement, logical 6-19

APP. I

COM1: 6-24
COM2: 6-24
concatenation 6-23
constants 6-7
CONT Command 7-38
converting
 ASCII code to character
  7-21
 character to ASCII code 7-5
 from number to string 7-141
 from numeric to octal
 hexadecimal 7-144, 7-92
 one numeric precision to
  another 7-22, 7-41
 string to numeric 7-43
converting programs to
 Panasonic BASIC E-1
coordinates
 specifying 6-27
COS Function 7-40
cosine 7-40
CSNG Function 7-41
CSRLIN Variable 7-42
Ctrl 3-5
Ctrl-Break 3-6
Ctrl-Num Lock 3-6
current directory 5-5
cursor 4-4
cursor control keys 4-4
Cursor Down key 4-4
Cursor Left key 4-5
cursor position 4-4
Cursor Right key 4-5
Cursor Up key 4-4
CVI, CVS, CVD
 Functions 7-43

D

Data Segment D-10
DATA Statement 7-44
DATE$ Statement
 and Variable 7-46
DEBUG D-7
declaring variable types 6-11
defining arrays 6-12
DEFDBL 6-7
DEF FN Statement 7-48
DEF SEG Statement 7-50
DEF USR Statement 7-54
DEFtype (-INT, -SNG,
 -DBL, -STR) 7-52
Del key 4-7
DELETE Command 7-55
deleting a file 7-112
deleting a program 7-143
deleting arrays 7-71
deleting characters 4-7
deleting program lines 7-55
delimiting reserved words 6-3
device name 5-4
DIM Statement 7-56
direct mode 2-15
display program lines 7-122
division 6-15
DOS prompt 2-7
double precision 6-5
DRAW Statement 7-58
DS (BASIC's Data Segment)
 D-10

E

EDIT Command 7-65
editor 4-2
editor keys
 Backspace 4-8
 Ctrl-Break 4-8
 Ctrl-End 4-7
 Ctrl-Home 4-4
 Cursor Down 4-4
  Cursor Left 4-5

APP. I

Cursor Right 4-5
Cursor Up 4-4
Del 4-7
End 4-6
Esc 4-8
Ins 4-7
Next Word 4-5
Previous Word 4-6
Tab 4-6
ELSE 7-93
End key 4-6
end of file 7-70
END Statement 7-66
ending BASIC 7-257
ENVIRON Statement 7- 67
ENVIRON$ Function 7-69
Enter key 3-4
entering BASIC program 2-7
EOF Function 7-70
equivalence 6-21
EQV 6-21
ERASE Statement 7-71
erasing a file 7-112
erasing a program 7-143
erasing arrays 7-71
erasing characters 4-7
erasing part of a line 7-55
erasing program lines 7-55
ERDEV and ERDEV$
Varicables 7-72
ERR and ERL Variables
7-73
error messages Appendix A
A-1
error number A-19
ERROR Statement 7- 75
error trapping 7-147
Esc key 4-8
event trapping
COM(n) 7-36
ON KEY(n) 7-151
STRIG(n)
(joy stick button) 7-254
exclusive or 6-21
executable statements 2-17
executing a program 4-11
EXP Function 7-76
exponential function 7-76
exponentiation 6-15
expressions 6-14
extended codes C-5
extension, filename 5-4

F

false 6-19
FIELD Statement 7-77
file specification 5-4
filename 5-4
filename extension 5- 4
files
naming 5-4
opening 5-9
position of 7-126
size 7-129
FILES Command 7-79
FIX Function 7-81
fixed point 6-4
floating point 6-4
FOR and NEXT
Statements 7-82
foreground 7-32
formatting math output 7-197
FRE Function 7-85
frequency table 7-243
Function keys 3-3
Functions 2-19

G

Game Control Adapter 6-28
garbage collection 7-85
GET Statement (Files) 7- 86
GET Statement (Graphics)
7-87

GOSUB and RETURN
 Statements 7-89
GOTO Statement 7-91
graphics 6-25
graphic mode 6-26
graphics statements
 CIRCLE 7-23
 COLOR 7-34
 DRAW 7-58
 GET 7-87
 LINE 7-116
 PAINT 7-175
 POINT Function 7-190
 PSET and PRESET 7-205
 PUT 7-207

H

hard copy of screen 3-6
HEX$ Function 7-92
hexadecimal 6-5
high resolution 6-26
Home key 4-4
how to format output 7-197

I

IF Statement 7-93
IMP 6-22
implication 6-22
indirect mode 2-15
INKEY$ Variable 7-96
INP Function 7-98
INPUT# Statement 7-101
INPUT Statement 7-99
INPUT$ Function 7- 103
Ins key 4-7
insert mode 4-7
inserting characters 4-7
INSTR Function 7-105
INT Function 7-106
integer
 converting to 7-22
integer division 6-15

J

joy stick 6-28
joy stick button 6-28

K

KEY Statement 7-107
KEY (n) Statement 7-111
keywords 2-18
KILL Command 7-112
KYBD: 6-24

L

LEFT$ Function 7-113
LEN Function 7-114
length of file 7-129
length of string 7- 114
LET Statement 7-115
LINE Statement 7-116
LINE INPUT#
 Statement 7-120
LINE INPUT Statement
 7-119
lines
 drawing in graphics 7-116
 format 2-16
 line numbers 2-16

APP. I

LIST Command 7-122
list program lines 7-122
listing files on disk 7-79
LLIST Command 7-124
LOAD Command 7-125
loading binary data 7-125
LOC Function 7- 126
LOCATE Statement 7-127
LOF Function 7-129
LOG Function 7-130
logarithm 7-130
logical operators 6-19
LPOS Function 7-131
LPRINT and LPRINT USING Statements 7-132
LPT1: 6-24
LPT2: 6-24
LPT3. 6-24
LSET and RSET Statements 7-134

M

machine language subroutines D-1
math output, formatting 7-197
medium resolution 6-26
MERGE Command 7-136
MID$ Function and Statement 7- 137
MKDIR Statement 7-139
MKI$, MKS$, MKD$ Functions 7-141
MOD 6-15
modulo arithmetic 6-15
multiple statements on a line 2-17
multiplications 6-15

N

NAME Command 7-142
naming files 5-4
negation 6-15
NEW Command 7-143
NEXT 7-82
Next Word 4-5
nonexecutable statements 2-17
NOT 6-19
Num Lock 3-7
number pad 3-7
numeric characters 2-20
numeric comparisons 6-17
numeric constants 6- 9
numeric expressions 6-14
numeric precision 6-5
numeric representation 6-4
numeric variables 6-10

O

OCT$ Function 7-144
octal 6-5
Ok prompt 2-7
ON COM(n) Statement 7-145
ON ERROR Statement 7-147
ON KEY(n) Statement 7-151
ON PLAY(n) Statement 7-154
ON STRIG(n) Statement 7-156
ON TIMER(n) Statement 7-158
ON-GOSUB and ON-GOTO Statements 7-149
OPEN "COM... Statement 7-165
OPEN Statement 7-160
operation modes 2-15
operators
arithmetic 6- 15

logical 6-19
relational 6-16
string 6-23
OPTION BASE
Statement 7-172
options on BASIC
Command line 2-8
OR 6-20
OUT Statement 7-173
output, formatting 7-197

P

PAINT Statement 7-175
palette 7-34
Path/file access error A-16
pathnames 5-5
Path not found A-16
PEEK Function 7-181
PLAY Statement 7-182
PLAY Statement
(ON, OFF and STOP)
7-186
PLAY(n) Function 7-187
PMAP Function 7- 188
POINT Function 7-190
POKE Statement 7-192
POS Function 7-193
Previous Word 4-6
PRINT# and PRINT#
USING Statements 7-203
print screen 3-6
PRINT Statement 7-194
PRINT USING
Statement 7-197
program editor 4-2
PrtSc 3-6
PSET and PRESET
Statements 7-205
PUT Statement (Files) 7-206
PUT Statement
(Graphics) 7-207

R

random files 5-13
random numbers 7-226
RANDOMIZE Statement
7-211
READ Statement 7-213
record length 7- 161
redirection of standard I/O 2-13
?Redo from start 7-100
relational operators 6-16
relative form for specifying
coordinates 6-27
REM Statement 7-215
remarks 7-215
renaming files 7-142
RENUM Command 7-216
reserved words 6-3
RESET Command 7-218
RESTORE Statement 7-219
RESUME Statement 7-220
RETURN Statement 7-222
RIGHT$ Function 7-223
RMDIR Statement 7- 224
RND Function 7-226
root directory 5-5
RUN Command 7-228
running a program 4-11

S

SAVE Command 7-230
saving binary data 7-230
SCREEN Function 7-232
SCREEN Statement 7-234
SCRN: 6-24
seeding random number
generator 7-212
sequential files 5-9
SGN Function 7-237
SHELL Statement 7-238
Shift-PrtSc 3-6
SIN Function 7-241
sine 7- 241

single precision 6-5
soft keys 7-107
SOUND Statement 7-242
sounds 7-242
SPACE$ Function 7-245
SPC Function 7-246
special characters 2-20
specification of files 5-4
specifying
 coordinates 6-27
SQR Function 7-247
square root 7-247
Statements 2-18
STICK Function 7-248
STOP Statement 7-249
STR$ Function 7-251
STRIG Statement and
 Function 7-252
STRIG(n) Statement 7-254
string comparisons 6-17
string constants 6-9
string expressions 6-14
string variables 6-10
STRING$ Function 7-255
subroutines 7-89
subroutines, machine
 language D-1
subtraction 6-15
SWAP Statement 7-256
syntax errors 4-11
SYSTEM Command 7-257
System Reset 3-6

T

TAB Function 7-258
Tab key 4-6
TAN Function 7-259
tangent 7-259
tempo table 7-244
terminating BASIC 7-66
text mode 6-26
THEN 7-93
TIMER Function 7-260
TIMER Statement 7-261
TIME$ Variable and
 Statement 7-262
trace 7-264
trigonometric functions
 arctangent 7-6
 cosine 7-40
 sine 7-241
 tangent 7-259
TRON and TROFF
 Commands 7-264
true 6-19
truncation 6-15
truncation of program
 lines 7-55
typewriter keyboard 3-4

U

user-defined functions 7-48
USR Function 7-265

V

VAL Function 7-266
Variables 2-19
VARPTR Function 7-267
VARPTR$ Function 7-269
VIEW PRINT Statement
 7-270
VIEW Statement 7-271

W

WAIT Statement 7-274
WHILE and WEND
 Statements 7-276
WIDTH Statement 7-278
Window and View 6-27
WINDOW Statement 7-281
workspace 2-9
WRITE# Statement 7-287
WRITE Statement 7-286

X

XOR 6-21

USA

Panasonic Industrial Company
Division of Matsushita Electric Corporation of America
One Panasonic Way,
Secaucus, New Jersey 07094

Panasonic Hawaii Inc.
91-238 Kauhi St. Ewa Beach
P.O. Box 774
Honolulu, Hawaii 96808-0774

Panasonic Sales Company
Division of Matsushita Electric of Puerto Rico, Inc.
Ave. 65 De Infanteria, KM 9.7
Victoria Industrial Park
Carolina, Puerto Rico 00630

CANADA

Matsushita Electric of Canada Limited
5770 Ambler Drive, Mississauga,
Ontario L4W 2T3

OTHERS

Matsushita Electric Trading Co., Ltd.
32nd floor, World Trade Center Bldg.,
No. 4-1, Hamamatsu-Cho 2-Chome,
Minato-Ku, Tokyo 105, Japan
Tokyo Branch P.O. Box 18 Trade Center

Printed in Japan RQX4307ZD F0184T0