VS

BASIC Language Reference

WANG

BASIC Language Reference

3rd Edition
July, 1980
© Wang Laboratories, Inc. 1979
800-1202BA-03

WANG LABORATORIES, INC.

ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

This manual replaces and obsoletes the second edition of *VS BASIC Language Reference* (800-1202BA-02). For a list of changes made to this manual since the previous edition, see the "Summary of Changes."

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

PREFACE

This manual is designed as a reference for Wang VS BASIC Version 3.2. For information on the VS Operating System see VS Operating System Services (800-1107OS). For a detailed discussion of the programming utilities see VS Programmer's Introduction (800-1101PI).

The manual is divided into two parts. Part 1 contains general discussions of the form of programs and data, and of the use of the different types of VS BASIC statements. These discussions generally assume minimal programming knowledge.

Part 2 contains the specific syntax for each VS BASIC instruction. In most cases clarifying examples are provided, along with details of the required formats.

VS BASIC was originally based upon 2200 BASIC; some important lexical and I/O differences do exist as explained in Chapters 7 and 8.

Wang VS BASIC Version 3.2 must have Release 4.0 (or higher) of the VS Operating System in order to perform properly. If the user tries to run a VS BASIC Version 3.2 program on an earlier version of the Operating System, an error message informing the user that the correct version of the Operating System is not being used, will be issued. In addition, along with the new compiler a new EDITOR will be provided. This new version of the EDITOR must be used with Wang VS BASIC Version 3.2.

SUMMARY OF CHANGES

FOR THE 2nd EDITION OF THE VS BASIC LANGUAGE REFERENCE MANUAL

TOPIC	DESCRIPTION		PAGES
Release 3.2	Pre-release 3.2	Release 3.2	
Syntax Changes		Required spacing within a statement	14,15
	One or two character variable names	Long variable names	18
	#PI intrinsic function	PI intrinsic function	35,214,235
	SELECT D	SELECT DEGREES	36,256
	SELECT G	SELECT GRADS	36,256
	SELECT R	SELECT RADIANS	36,256
		Statement labels	62,63
		FILESEQ for tape files	100,220-223
		CLOSE WS	140
		CLOSE CRT	140
	CONVERT X TO Y$,(###)	CONVERT X TO Y$,PIC(###)	142
	FMT PD (6.4)	FMT PD (6.4)	152-154
	Non-numeric file, library, and volume name only	Numeric file, library, and volume name	220
	PACK (###)	PACK PIC (###)	225,226
		SELECT WS	256
	SELECT P	SELECT PAUSE	256
	TRAN [R]	TRAN [REPLACING]	273
	UNPACK (###)	UNPACK PIC (###)	274
General	Miscellaneous	editorial changes	

SUMMARY OF CHANGES
FOR THE 3rd EDITION OF <u>VS BASIC LANGUAGE REFERENCE</u>

TOPIC	DESCRIPTION	PAGES
PRINT Files	CLOSE PRINTER SELECT PRINTER	140 140, 256
Miscellaneous	Technical and Editorial	220, 253, 256, 266, 277

TABLE OF CONTENTS

PART 1 INTRODUCTION TO BASIC

CHAPTER 1 INTRODUCTORY CONCEPTS

1.1 An Overview: BASIC on the Wang VS 1
1.2 Communicating with the VS 2
The Workstation 2
Use of PF Keys: Menus 3
Logging On 3
The Command Processor 3
1.3 The VS Operating System 3
The Data Management System (DMS) 4
File Hierarchy 4
1.4 BASIC Program Development 5
The EDITOR 6
The BASIC Compiler 8
The LINKER Utility 10
Running the Object Program 11

CHAPTER 2 PROGRAM FORMAT

2.1 Introduction 12
2.2 Statements 12
2.3 Line Format 13
Spacing 14
Multiple Statement Lines 15
Continuation of Statements 15
Sequence of Execution 16
2.4 Program Documentation 16
Comments 16
Compiler Directives 17

CHAPTER 3 DATA FORMATS

3.1 Introduction 18
3.2 Constants, Variables, Receivers, and Expressions 18
3.3 Numeric Data 19
Floating-point Constants 19
Integer Constants 21
Numeric Variables 21
Floating Point and Integer Calculation 22
3.4 Alphanumeric Data 23
Literals (Alphanumeric Constants) 23
Alphanumeric Variables 24
3.5 Array Variables 26
One-Dimensional and Two-Dimensional Arrays 27
Dimensioning an Array 29

CHAPTER 4 NUMERIC OPERATIONS

4.1 Introduction 31
4.2 Numeric Operators 31
The Assignment Operator 31
Arithmetic Operators 32
Relational Operators 33
4.3 Numeric Expressions 34
4.4 Numeric Functions 34
Intrinsic Functions 35
User-defined Functions 40
4.5 Mixed Mode Arithmetic 41
4.6 Summary of Numeric Data Types and Terms 41
Floating-Point Data 41
Integer Data 42
Numeric Terms 43

CHAPTER 5 ALPHANUMERIC OPERATIONS

5.1 Introduction 45
5.2 Alphanumeric Operators 45
The Assignment Operator 45
The Concatenation Operator 46
Relational Operators 47
5.3 Alpha Array Strings 47
5.4 Alpha Expressions and Alpha Receivers 48
Alpha Expressions 48
Alpha Receivers 48
5.5 Alphanumeric Functions 49
5.6 Numeric Functions with alpha Arguments 50
LEN 50
NUM 51
POS 51
VAL 52
5.7 Logical Expressions 53
Evaluation of Logical Expressions 54
Logical Operators 55
5.8 Summary of Alphanumeric Data Forms and Terms 56
Alphanumeric Length 56
Alphanumeric Terms 57
Alphanumeric Operations 59

CHAPTER 6 CONTROL STATEMENTS

6.1 Introduction 61
6.2 Statement Labels 62
6.3 Subroutines 63
6.4 Internal subroutines 64
GOSUB Subroutines 64
GOSUB' Subroutines 65
Program Function Keys 66

6.5 External Subroutines 67
Operation of External Subroutins 67
Form of External Subroutine Calls and Definitions 68
Compiling, Linking, and Running 69
Passing Values to External Subroutines 69
Initialization of Subroutine Variables 73
Argument Types 74
Use of External Subroutines 77

CHAPTER 7 WORKSTATION AND PRINTER INPUT/OUTPUT

7.1 Introduction 78
Output 78
Input 79
7.2 Printer Output 79
Expanded Print 80
7.3 Workstation Input/Output 80
Wraparound 80
Scrolling 81
Field Attribute Characters (FACs) 81
7.4 The USING Clause and Format Control Statements 82
The FMT Statement 83
The Image (%) Statements 83
Use of FMT and Image (%) Statements 84
7.5 The ACCEPT Statement 85
Screen Formatting 86
Data Entry and Validation 88
PF Key Usage and Program Branching 89
Summary of ACCEPT Execution 91
7.6 The DISPLAY Statement 91
7.7 Workstation Programming Considerations 92

CHAPTER 8 FILE INPUT/OUTPUT

8.1 Introduction 93
8.2 Files 93
File Types 93
Record Types: Length and Compression 95
8.3 Use of Files by BASIC Programs 95
The SELECT Statement 96
The OPEN and CLOSE Statements 98
File I/O Modes 101
File I/O Buffering and the Record Area 101
8.4 The File I/O Statements 103
The READ Statement 103
The GET Statement 103
The WRITE Statement 104
The PUT Statement 104
The REWRITE Statement 104
Summary of Data Flow Controlled by File I/O Statements . 105
Data Representation in File I/O 105

8.5 Intrinsic File I/O Functions 105
FS (filexpression) 106
KEY (filexpression [,exp]) 106
MASK (filexpression) 107
SIZE (filexpression) 107
8.6 Error Recovery 107
8.7 Examples of File I/O 109

CHAPTER 9 DATA CONVERSION AND MATRIX STATEMENTS

9.1 Data Conversion Statements 113
9.2 Matrix Statements 113
Matrix I/O Statements 114
Matrix Assignment Statement 114
Matrix Arithmetic and Sorting Statements 114
Array Dimensioning 115
Matrix Statement Rules 116

PART 2 VS BASIC STATEMENTS AND FUNCTIONS

ABC Function 119
ACCEPT Statement 120
ADD[C] Logical Operator 126
ALL Function 128
AND Logical Operator 129
ARCCOS Function 130
ARCSIN Function 131
ARCTAN Function 132
ATN Function 133
BIN Function 134
BOOLh Logical Operator 135
CALL Statement 137
CLOSE Statement 140
COM Statement 141
CONVERT Statement 142
COPY Statement 144
COS Function 145
DATA Statement 146
DATE Function 147
DEF Statement 148
DEF FN' Statement 150
DELETE Statement 154
DIM Statement 155
DIM Function 156
DISPLAY Statement 157
EJECT Compiler Directions 158
END Statement 159
EXP Function 160
FMT Statement 161
FOR Statement 164
FORM Statement 165
FS Function 166

GET Statement 167
GOSUB Statement 168
GOSUB' Statement 169
GOTO Statement 170
HEX Function 171
HEXPACK Statement 172
HEXPRINT Statement 175
HEXUNPACK Statement 176
IF... THEN... ELSE Statement 177
Image (%) Statement 179
INIT Statement 181
INPUT Statement 182
INT Function 185
KEY Function 186
LEN Function 187
LET Statement 188
LGT Function 190
LOG Function 191
MASK Function 192
MAT + (MAT addition) Statement 193
MAT ASORT/DSORT Statement 194
MAT CON (MAT CONstant) Statement 196
MAT= (MAT assignment) Statement 197
MAT IDN (MAT identity) Statement 198
MAT INPUT Statement 199
MAT INV (MAT inverse) Statement 201
MAT * (MAT multiplication) Statement 203
MAT PRINT Statement 204
MAT READ Statement 205
MAT REDIM Statement 206
MAT()* (MAT scalar multiplication) Statement 207
MAT - (MAT subtraction) Statement 208
MAT TRN (transpose) Statement 209
MAT ZER (MAT ZERO) Statement 210
MAX Function 211
Mathematical Functions 212
MIN Function 215
MOD Function 216
NEXT Statement 217
NUM Function 218
ON Statement 219
OPEN Statement 220
OR Logical Operator 224
PACK 225
$PACK/$UNPACK Statement 227
PI Intrinsic Constant 235
POS Function 237
PRINT Statement 238
PUT Statement 241
READ Statement 242
READ File Statement 243
REM[ARK] Statement 245
RESTORE Statement 246
RETURN Statement 247

RETURN CLEAR Statement 248
REWRITE Statement 249
RND Function 251
ROTATE [C] Statement 252
ROUND Function 253
SEARCH Statement 254
SELECT Statement 256
SELECT File Statement 257
SGN Function 260
SIN Function 261
SIZE Function 262
SKIP Statement 263
SQR Function 264
STOP Statement 265
STR Function 266
SUB Statement 267
TAN Function 270
TIME Function 271
TITLE Statement 272
TRAN Statement 273
UNPACK Statement 274
$UNPACK Statement 275
VAL Function 276
WRITE Statement 277
XOR Statement 279

APPENDICES

Appendix A VS BASIC Reserved Words 280
Appendix B VS BASIC Compiler Options 282
Appendix C Numeric Data Representation in VS BASIC: Hexadecimal/ Decimal Conversion Errors 285
Appendix D Floating-Point and Integer Calculations 286
Appendix E Numeric Data Format Compatibility between VS BASIC and COBOL 289
Appendix F VS Character Set 291
Appendix G VS Field Attrubite Characters 292
Appendix H ASCII Collating Sequence 293
Appendix I VS BASIC Error Messages 294
Appendix J CVBASIC User Aid (Conversion from BASIC 2.3 to 3.2) 300

INDEX 302

CUSTOMER COMMENT FORM . . . . Last Page

PART 1
INTRODUCTION TO BASIC

CHAPTER 1
INTRODUCTORY CONCEPTS

1.1 AN OVERVIEW: BASIC ON THE WANG VS

Wang VS BASIC is a compiled, general-purpose, high-level programming language developed by Wang Laboratories for use on the VS System. This modified version of the original Dartmouth BASIC offers all the original language's important features, as well as added capabilities which suit it for both technical and commercial applications. Although VS BASIC is extremely powerful and versatile, it is also easily learned by beginning programmers because:

1. BASIC statements bear a close resemblance to the English language, giving beginning programmers clues to the BASIC meaning. In situations where formulae must be used, the BASIC language resembles standard algebraic notation and other programming languages such as FORTRAN.

2. A programmer does not need to know much about BASIC to write a simple program. The programmer need not learn about the advanced capabilities of BASIC until a specific need for those capabilities arises.

VS BASIC Version 3.2 incorporates diverse features that aid in program development and increase data processing versatility, including:

Variable names up to 64 characters long.* Long variable names enable the programmer to assign mnemonic and self-explanatory names. Programs using such variable names are easier to read and debug than are the limited two-character names found in most BASIC implementations.

Alphanumeric statement labels.* Any statement in a VS BASIC program may be identified by an arbitrary statement label up to 64 characters long, which can be referenced in any program branch statement (GOTO, IF...THEN...ELSE, etc.). Programs can thus be written without regard to line numbers, as is necessary in most BASIC implementations. Blocks of program code can be given mnemonic labels which indicate their function, again increasing program readability and ease of debugging.

* These features are new to Version 3 of VS BASIC.

Workstation, file, and printer I/O statements. The ACCEPT and DISPLAY statements enable BASIC programs to make full use of the capabilities of the VS workstation, allowing sophisticated screen formatting and use of Program Function (PF) keys for data entry and program control. The FMT and Image (%) statements allow precise control over format of file and printer I/O.

Integer and floating-point formats. Numeric data can be stored and manipulated in either format. Use of integer format can increase both the speed and efficiency of memory use.

Alphanumeric operations. Extensive facilities for the manipulation of alphanumeric data are provided. Substrings can be extracted from strings of characters, and strings can be concatenated (put together) or searched for particular substrings.

Boolean logic functions on binary values. All 16 Boolean functions of two variables are available in VS BASIC. Results can be used in alphanumeric expressions or output as hexadecimal numbers or ASCII character strings.

Intrinsic and user-defined functions. A full set of arithmetic and trigonometric functions is provided by VS BASIC. In addition, the programmer can define and name any arbitrary numeric function to be used in a program.

Multilingual subroutines. Programs written in VS BASIC can call subroutines written in other languages (e.g., COBOL, Assembler) and vice versa.

1.2 COMMUNICATING WITH THE VS

1.2.1 The Workstation

The principal means of user communication with the VS is through the VS workstation. The workstation is a terminal consisting of a Cathode Ray Tube (CRT) display screen and a typewriter-like keyboard. The screen displays output from the computer and text typed by the user on the keyboard.

In addition to the keys which correspond to the alphabetic and numeric characters that appear on the screen, the keyboard has 16 Program Function (PF) keys. By using the SHIFT Key, a total of 32 PF key values can be obtained.

Whenever the workstation is ready to accept input from either the keyboard or the PF keys, a cursor is shown on the screen. The cursor appears as a flashing bar under the character position where the next character typed will appear. The cursor (and thus the position of the next character) can be moved with the four cursor control keys, each of which is marked with an arrow indicating the direction in which it moves the cursor.

At any particular time, certain keys may be accepted for input, while others are not. For example, a program may prompt the user to input certain numeric data. In this case, the use of the alphabetic keys is invalid. Any time an invalid key is pressed, either from the keyboard or the PF keys, the workstation emits a beep (the workstation alarm), and the key is ignored.

1.2.2 Use of PF Keys: Menus

Most commands and options entered by the user to system programs are entered by means of the PF keys in response to menus. A menu is a list of possible commands or options displayed on the workstation screen by a program. Next to the description of each command is the number of one of the PF keys. Commands are selected by pressing the appropriate PF key.

Communication through PF key response to menu screens is extensively used in VS system programs since it frees the user from the necessity of typing many commands and remembering their syntactical arrangements. This enables programs to be highly interactive and "self-documenting."

PF keys can also be used by BASIC programs to control the sequence of program execution and to assign values to variables in the program (see Subsection 6.4.3, Program Function Keys, and Subsection 7.5.3, PF Key Usage and Program Branching).

1.2.3 Logging On

Before one can use the VS system, one must log on to the system by entering a valid user ID and password at the workstation. User IDs and passwords are assigned to authorized users by the system security administrator at each VS installation. When the logon procedure is completed, the Command Processor menu is displayed.

1.2.4 The Command Processor

The Command Processor is the program which runs whenever no other system or user program is executing. When a user first logs on and whenever any program is completed (or interrupted with the HELP key), the Command Processor Menu is displayed on the workstation screen. From this menu, the user can run a program; examine and manage files, libraries, and volumes (see Subsection 1.3.2, File Hierarchy); examine the status of peripheral devices; or perform a variety of other functions.

1.3 THE VS OPERATING SYSTEM

The VS Operating System consists of a set of programs which manage the hardware and software resources of the VS. The Operating System allocates processor time and memory space to user tasks, processes all input/output operations between user programs and disk or tape files, and maintains a security system to ensure that only authorized users can gain access to the system hardware, software, and data. The Operating System also includes the Command Processor, language compilers (e.g., BASIC and COBOL), such program development aids as the EDITOR and LINKER, file management utilities, and various other utility programs. The programs which are supplied as part of the Operating System are called system programs (as distinguished from those written by users, called user programs).

1.3.1 The Data Management System (DMS)

The Data Management System (DMS) consists of several programs which are part of the VS Operating System, and which processes all input/output transactions between user or system programs and data stored in files on magnetic disk or tape. DMS also controls the creation of new files. The operation of DMS is transparent in that the user does not directly interact with DMS. When a user program is running and needs to perform some file I/O operation, DMS is automatically called to perform the necessary operations; the user program then continues executing with no direct involvement in the I/O operation. Many of the functions performed by DMS involve the complex internal housekeeping tasks required to insure that information stored in files remains properly organized for reliable and efficient access through all input/output operations.

1.3.2 File Hierarchy

A file is a collection of data stored on either magnetic disk or tape, and identified by a file name. Groups of disk files are organized into a hierarchical structure with two higher levels: libraries and volumes. Groups of tape files are organized into volumes (there are no tape libraries).

The most comprehensive unit in the file management hierarchy is the volume. A volume is an independent physical storage medium, such as a diskette, disk pack, or tape. The volume name provides a device-independent means of identifying physical storage units. Once a diskette, disk pack, or tape has been assigned a volume name, it can be mounted at any available drive unit and accessed by name, without reference to the address or physical characteristics of the disk or tape unit itself.

Immediately below the volume in the disk hierarchy is the library. A volume may contain one or more user libraries, but a single library may not continue onto a second volume. Each library contains one or more files (every disk file must be assigned to a library). The VS places no particular restrictions on the types of files placed in a library; a single library may be used for program and data files, or special libraries may be designated for each file type. The conventions governing library usage are completely determined at each individual installation, based on its particular needs and standards.

Duplicate file names cannot be used within the same library, but they may be used in different libraries. Similarly, duplicate library names are not permitted on the same volume, but may be used on separate volumes. Duplicate volume names are allowed but not recommended.

File and library names can contain up to eight characters. Volume names contain up to six characters. Each name must begin with an uppercase letter, a number, or one of the special characters $, #, or @; subsequent characters may be any alphanumeric character, including the special characters. Embedded spaces are not allowed.

1.4 BASIC PROGRAM DEVELOPMENT

The VS Central Processing Unit (CPU) hardware, like most digital computers, can directly execute only instructions written in machine language. Machine language consists of groups of electrical impulses represented as binary or hexadecimal (base 16) numbers. Machine language is cumbersome for programmers, and using it to program directly is tedious.

VS BASIC, on the other hand, is an extremely convenient and readable language in which to write programs, but programs written in BASIC are not directly executable by the CPU. In order for a BASIC program to be executed (or "run"), it must first be translated into machine language. This translation is accomplished by a large program called the BASIC compiler; the translation process is called compilation.

The VS BASIC compiler takes as input a file containing a program written in the VS BASIC language as described in this manual. Such a program is called a source program; the file containing it is a source file. As output, the compiler produces a file containing the machine language translation of the source program. This machine language program is called an object program; it is contained in an object file. The object program can be run using the RUN command (PF 1) of the Command Processor.

Development and execution of a VS BASIC program thus consists of three steps (not including the logical design and coding of a program into BASIC instructions):

1. The BASIC source program is entered from the workstation using the EDITOR utility and stored in the source file.

2. The source program is compiled to produce an object program by the BASIC compiler, and stored in the object file.

3. The object program is run from the Command Processor.

These steps can be performed separately by running first the EDITOR, then the BASIC compiler, and finally the user's object program, returning to the Command Processor after each step. The entire process can also be performed from the EDITOR, enabling the user to compile and run programs directly from its Special Menu. The EDITOR is described in detail in Subsection 1.4.1, the EDITOR, and in the VS Programmer's Introduction; the process of creating and running a new BASIC program is summarized in Subsections 1.4.1, The EDITOR, and 1.4.2, The BASIC Compiler.

1.4.1 The EDITOR

To run the EDITOR, invoke the RUN command from the Command Processor Menu (PF 1), type EDITOR for the program name, and key ENTER.

The EDITOR first displays an Input Definition screen, requesting the following information:

LANGUAGE -- Type B or the word BASIC.

FILE, LIBRARY, VOLUME -- If a new file is to be created, leave the file name blank. Names are assigned to new files after the text of the file has been entered, with the CREATE command (PF 5). ~~I~~f an existing file is to be edited, enter its name, and the names of the library and volume on which it is contained. LIBRARY and VOLUME may have default values set when this screen appears. These can be changed by simply typing over the defaults.

LINES -- If significantly more than 500 lines are to be added to a file in this session, enter an estimate of the number of lines to be added.

When all of this information has been typed in appropriately, key ENTER.

The EDITOR next creates a work file for text editing. The editing of source text actually takes place in this temporary work file. In order to permanently store any text entered in the EDITOR, the user must either create a new file of the edited text or, if an old file was used, replace the old text with the edited text. The original file is not altered until a replace is done, as all changes are made in the work file. Files are created and replaced with the CREATE (PF 5) and REPLACE (PF 6) commands from the EDITOR's Special Menu.

The EDITOR then displays its normal menu, which contains 14 functions for examining, entering, and editing source text. The most important functions are briefly explained here. More detail on these and explanations of the other functions may be found in the VS Programmer's Introduction.

PF 1 - DISPLAY -- Display mode displays the user's file on the screen. The first time this command is used in an EDITOR session on an existing file, the file is displayed starting with the first line of text. Subsequent uses of this command return to displaying the file at the point where the last editing function was performed. While in display mode, different portions of the file may be examined by using PF keys 2 through 6. PF 1 is equivalent to PF 9 when there are no lines in a file (i.e., when a new file is first created).

PF 9 - MOD -- Modify mode allows the user to enter a new program, modify existing source lines, or add lines to the end of an existing program.

PF 11 - INS -- Insert mode allows text to be inserted in an existing program between lines, before the beginning of the program, or at the end. Unlike the modifiy mode, the line numbers supplied by the EDITOR can be altered in place, if the user wishes. Before pressing PF 11, the cursor should be positioned on the line after which the new line is to be inserted.

PF 12 - DEL -- Delete mode allows the user to delete text -- either a specific line, a range of lines, or all lines -- from the source file. Before pressing PF 12, the cursor should be positioned on the first line to be deleted.

PF 16 - MENU -- Activates the EDITOR's Special Menu.

To enter lines of text for a new file, enter either modify (PF 9) or insert (PF 11) mode and simply type in the lines. Pressing ENTER sends the lines which were just typed to the system for processing. This must be done after every inserted line. In modify mode, the screen may be filled with new lines before ENTER is keyed.

In order to return to display mode from modify or insert modes, press PF 1 after the last line of text is ENTERed (or, if in modify mode, press ENTER after typing in no new lines of text).

When the entire BASIC program has been entered, it can be stored into a disk file, compiled, or run directly. All of these functions are performed from the EDITOR's Special Menu. The Special Menu is obtained by pressing PF 16 from display mode.

The Special Menu has thirteen functions. The most important ones are listed below. These functions, as well as those not described here, are described in detail in the VS Programmer's Introduction.

PF 1 - DISPLAY -- The EDITOR is returned to the point in text editing from which the Special Menu was invoked.

PF 5 - CREATE -- A new file of the edited text is generated. The user is asked to supply file, library, and volume names and several optional pieces of information, including a retention period during which the file cannot be scratched.

PF 6 - REPLACE -- The old input file is replaced with the new edited text.

PF 9 - RUN -- An uncompiled program is compiled and run, or a compiled program is run. If the text has not already been successfully compiled in this EDITOR session since the last text entry was made, RUN invokes the BASIC compiler and LINKER to compile the program, and then automatically runs the program (unless there are serious compilation errors). If compilation is not necessary, the program is run.

PF 10 - COMPILE -- The BASIC compiler and (optionally) the LINKER utility are invoked, but the program is not actually run.

PF 11 - ERRORS -- A list of detected errors is displayed. If the default value of ERRLIST in the Compiler/LINKER Options display was changed to NO, this list will not be displayed, and will not be accessible from the EDITOR.

PF 16 - EOJ -- EDITOR processing is ended and control is returned to the Command Processor.

NOTE:

The user must specify an object file name, library, and volume whenever a program is compiled from the EDITOR. Specifying a file name beginning with ## causes a temporary file to be created. Such a file is automatically scratched at the end of the EDITOR session.

1.4.2 The BASIC Compiler

The BASIC compiler can be invoked either from the Command Processor by the RUN (PF 1) command, or from the EDITOR by the RUN (PF 9) or COMPILE (PF 10) commands on the Special Menu. In either case, the compiler displays a list of options when it is invoked.

Options

The compiler options are described in detail in Appendix B. The three most important options are:

LOAD -- Directs the compiler to produce an object program as output. Its default value is YES. If NO is typed, no object program is produced. (The code generation phase of the compiler is not run.)

SOURCE -- Directs the compiler to produce a listing of the source code for the compiled program combined with a list of any compiler detected errors. YES causes the listing to be produced, while NO suppresses it. The default value is YES.

SYMB -- Directs the compiler to insert symbolic debug information into the object program, permitting subsequent use of the VS interactive symbolic debug facility when the program is run. Symbolic debug information can be removed from a program with the LINKER utility. The default value is YES.

When all desired options have been selected, key ENTER.

Input Definition

BASIC now requests the name of the source file to be used as input. Enter the file name, along with the appropriate library and volume names.

Output Definition

If LOAD = NO was specified, and if the program passes the compiler's syntax check with no error with severity equal to or greater than the specified STOP level (see Appendix B), a name for the output file to be created containing the compiled (object) program is requested. Enter the file name, along with the names of the library and volume to which it will be assigned. The following options may also be specified:

RECORDS -- The number of records in the output file is automatically determined by the compiler based on the size of the input file. In general, this value should not be changed by the user.

RETAIN -- During the specified retention period, the file cannot be scratched or renamed. Only the owner or a security administrator can change the retention period. If such protection is not deemed necessary, the RETAIN field should be left blank.

RELEASE -- If RELEASE=YES, any space originally allocated to the object file but not actually used is released for use by other files. Otherwise, the space remains reserved for use by the object file.

FILECLAS -- The object file may be assigned to one of the VS file protection classes. Consult the system security administrator to determine in which protection class a particular file belongs.

When the output file name and all options have been defined, key ENTER. The message BASIC COMPILATION OF PROGRAM X IN PROGRESS will appear on the screen while the compiler runs. When compilation is complete, control returns to either the Command Processor or the EDITOR, depending on how the compiler was initially invoked.

Return Code

When a compilation is completed, the first screen shown will specify a return code. The value of the return code indicates the severity of the errors found by the BASIC compiler in the source program. The possible return codes and their meanings are:

Code	Meaning
0	No errors.
4	Warning.
6 or 8	Severe error (program probably will not run correctly).
12 or 16	Terminal error (program will not run at all).

If production of the source listing was not suppressed, this listing and a list of compiler diagnostics (error messages) are printed on the selected printer, or directed to the print queue or the user's print library as specified by the user's PRNTMODE default (set with PF 2 from the Command Processor; see the VS Programmer's Introduction for an explanation). All other optional listings and tables are similarly printed, queued, or filed.

When the BASIC compiler is run from the EDITOR (by either the RUN (PF 9) or the COMPILE (PF 10) commands from the Special Menu), any error messages generated during the compilation can be viewed by keying PF 11 from the Special Menu.

1.4.3 The LINKER Utility

The VS LINKER is used to perform the following functions:

1. Link two or more object program modules or subroutines into a single executable program (see Section 6.5, External Subroutines).

2. Link library subroutines into a main program.

3. Remove symbolic debug information from an object program.

4. Replace one or more object program modules in a program.

The LINKER utility can be called whenever a program is compiled from the EDITOR. If the program is compiled using the BASIC compiler directly, the LINKER must be run independently by invoking the RUN command from the Command Processor and typing in LINKER as the program name. See the VS Programmer's Introduction for more information on the LINKER.

Note that due to changes in the Operating System, the user may not link BASIC Version 2.3 programs to BASIC Version 3.2 programs.

1.4.4 Running the Object Program

The compiled program is run with the RUN command from the Command Processor Menu. Press PF 1 to invoke this function, and type the BASIC object file name opposite PROGRAM. Type the appropriate library and volume names, and key ENTER to initiate execution of the program.

The program will continue to run until one of the following occurs:

1. An END statement is reached.
2. An "implied" END is reached because the physical end of the program is reached.
3. A fatal execution error occurs.
4. The user interrupts execution with the HELP key.

Any program can be interrupted at any time with the HELP key. A modified Command Processor Menu will be displayed. From this menu, the user can cancel or continue executing a program, or enter debug processing, as well as perform other system commands. The Debug Processor is a powerful tool used to detect hard-to-find errors in the logical design of a program. The Debug Processor is discussed in the VS Programmer's Introduction.

If a program completes execution without interruption by errors or the HELP key, control returns to either the Command Processor or the EDITOR, depending upon how execution of the program was initiated.

CHAPTER 2
PROGRAM FORMAT

2.1 INTRODUCTION

A VS BASIC source program consists of a series of instructions to the computer, called statements, which are written sequentially on numbered program lines. A program line may contain any number of statements. When a program is run, statements are executed sequentially in line number order. Multiple statements on the same line are executed left to right.

2.2 STATEMENTS

A statement usually begins with a word (called a "verb") which is typically an English verb, such as PRINT or INPUT. Following the verb is whatever information may be required to complete that particular statement. For example:

RETURN forms a complete statement by itself. It signals the end of a subroutine.

LET X=2 is an example of an assignment statement. In this case, the variable X is assigned a value of 2.

GOTO 40 transfers control to the given line number, and processing continues from there.

IF A=B THEN RETURN shows that another entire BASIC statement may follow the IF... verb. The IF statement causes some action to be taken depending upon whether or not a particular relation is true.

IF is a BASIC verb but is not a complete BASIC statement by itself.

Verbs form part of a larger set of reserved words. Reserved words are sequences of alphanumeric characters that have some predefined meaning to the BASIC compiler. Reserved words never contain any embedded spaces. Since reserved words and their meanings are built-in parts of the BASIC compiler, they cannot be used by the programmer as variable names or statement labels (see Section 6.2, Statement Labels). Appendix A contains a complete list of VS BASIC reserved words.

There are two types of BASIC statements: executable and non-executable. An executable statement specifies some action or a series of actions to be taken by the user's program at run time, such as assigning a value to a variable (LET statement), displaying or printing data on the workstation or printer (PRINT statement), or altering the order of program execution (GOTO statement). A non-executable statement provides information to the compiler at compilation time which may be required to generate the object program, such as the amount of storage to be allocated for certain variables (DIM statement) or the format to be used for printed output (FMT statement).

The following VS BASIC statements are defined as non-executable:

COM
DATA
DEF
DEF FN' or DEFFN'
DIM
EJECT
FMT or FORM
% (Image)
REM or *
SELECT, when used for file I/O (i.e., SELECT # and SELECT POOL; see Section 8.3.1, The SELECT Statement)
SUB
TITLE

2.3 LINE FORMAT

Each line in a VS BASIC program may be up to 72 characters long, including leading and embedded spaces (the workstation screen is 80 characters wide). Each character position is referred to by a column number, beginning with column 1 (the leftmost position). The first six columns of each line in a BASIC source file are reserved for a unique six-digit line number, leaving 66 columns (number 7-72) for program statements. Columns 73-80 may be used as a program identifier. Any line containing an asterisk (*) in column 7 is designated as a comment line and is ignored by the BASIC compiler (see Subsection 2.3.3, Continuation of Statements).

NOTE:

When the EDITOR is used to create or edit BASIC source files, program lines are displayed on the workstation screen with an extra space inserted between column 6 (which contains the rightmost digit of the line number) and column 7 (the first column available for typing program text characters). This extra space is also inserted to increase readability when the BASIC compiler prints source file listings. Thus, on printed listings and in this manual, the character in column 7 of a line actually appears in the 8th physical print position on the paper. This extra space is not, however, included in the internally stored representation of a program line.

2.3.1 Spacing

Within a statement, the VS BASIC compiler uses spaces between strings of nonblank characters to distinguish the significant entities or "tokens" which comprise the statement. To avoid ambiguity, it is important that spaces occur at certain places in a statement and do not occur at others. For example:

```
100 FOR K = I TO J            500 FORK = ITOJ
```

Both lines contain the same sequence of nonblank characters, and both are valid VS BASIC statements, but with completely different meanings. Line 100 is the beginning of a FOR...NEXT loop (see entries under FOR and NEXT, Part 2). In this statement, FOR and TO are VS BASIC reserved words (see Section 2.2, Statements) and K, I, and J are names of variables. Line 500 is an assignment statement (an "implied" LET statement) in which both FORK and ITOJ are variable names; the statement assigns the value of ITOJ to the variable FORK.

In general, spaces should occur in a statement so as to eliminate ambiguities in the interpretation of the statement. In particular, the following rules should be observed:

1. All VS BASIC reserved words, including verbs, must be spelled exactly as shown in Appendix A, with no embedded spaces. GOTO and GO TO are both valid and equivalent forms for the unconditional branch statement. GOSUB and GO SUB are also both valid and equivalent statements.

2. Literals (see Subsection 3.4.1, Literals (Alphanumeric Constants)) may contain any combination of blank and nonblank characters; a literal, however, cannot contain its delimiter.

3. No embedded spaces are allowed within variable names (see Subsection 3.3.3, Numeric Variables, for rules of forming variable names).

4. No embedded spaces are allowed within statement labels (see Section 6.2, Statement Labels, for rules governing formation of statement labels).

5. No embedded spaces are allowed in numbers (either line number references or constants).

6. One or more spaces is required between any reserved word, variable name, or statement label and any other reserved word, variable name, or statement label.

7. Spaces are ignored immediately before and after arithmetic operators (see Subsection 4.2.2, Arithmetic Operators), relational operators (see Subsection 4.2 3, Relational Operators), and punctuation marks.

2.3.2 Multiple Statement Lines

A program line may contain any number of statements. A line containing no statements is called a null line and consists simply of a line number followed by 74 spaces. If program line contains more than one statement, a colon (:) is used to separate one statement from the next, except following Image (%), TITLE, or EJECT statements (each of these statements is always considered as extending to the end of the line on which it occurs). For example:

```
400 LET TWEEDLEDUM=I : LET TWEEDLEDEE=J : LET ALICE$="CONFUSED"
```

A null statement may be inserted anywhere in a line by using one colon immediately after another, or two colons separated only by blanks.

2.3.3 Continuation of Statements

Statements may be continued beyond column 72 of a line by inserting an exclamation point (!) in column 72 of the line to be continued. For example:

```
400 LET ROCK=                                                   !
500 4
```

is equivalent to:

```
400 LET ROCK=4
```

Although a statement may begin on one line and end on another line, reserved words, constants, variable names (see Subsection 3.3.3, Numeric Variables), statement labels (see Subsection 6.2, Statement Labels) and line number references may not be split between lines. For example:

```
400 LE                                                          !
500 T ROCK =
```

is not a valid statement. Literal strings (see Subsection 3.4.1, Literals (Alphanumeric Constants)), however, may be split.

There is no limit to the number of lines which can be used to contain a single statement, nor to the number of statements which can occupy a single line.

2.3.4 Sequence of Execution

Execution of a BASIC program always proceeds in line number sequence from the lowest-numbered line through the highest-numbered line, unless the normal sequence of execution is altered by a program branch instruction. Program branch instructions include the following: FOR...NEXT loops, GOTO, GOSUB, GOSUB', CALL, RETURN, and, in certain cases, IF...THEN...ELSE. Program branch instructions are discussed more fully in Chapter 6 and Part II.

2.4 PROGRAM DOCUMENTATION

2.4.1 Comments

As an aid to program documentation, it is often useful to insert explanatory comments into the text of a program. Such comments must be distinguished in some way so that the compiler does not attempt to interpret them as executable program statements. VS BASIC provides three methods of inserting comments into programs.

1. Any line which has an asterisk (*) in column 7 (the first column following the six-digit line number) is treated as a comment line. The entire line is disregarded by the compiler and may contain any combination of printing characters. Comment lines of this form may not be continued (as described in Subsection 2.3.3, Continuation of Statements). Example:

2. Any statement beginning with the reserved word REM is treated as a comment (REMark). REM statements are ignored by the compiler and may appear wherever any other statement appears (see Section 2.3, Line Format). A REM statement may contain any combination of printing characters except a colon (:). A colon is considered to be a statement terminator and may be used to separate a REM statement from another statement on the same line. REM statements can be continued by the use of the exclamation point in column 72, as discussed above. Examples:

```
100 REM   CATASTROPHE THEORY SIMULATION OF CANINE BEHAVIOR

560 DIST=SIN(A)/COS(B) : REM CHECK FLAGS : IF FLAG1=1 THEN 1200
```

3. A comment may be inserted by enclosing it between the symbols "/*" and "*/". Comments delimited in this way (called "enclosed comments") may be inserted on a line alone, before, after, or between statements on a line, or within a statement. Enclosed comments within statements may occur before or after (but not within) reserved words, variable names, statement labels, line number references, numbers, literals, functions, operators and punctuation marks. All characters which follow the "/*" symbol (including subsequent occurences of "/*") are treated as part of the comment until the "*/" is encountered. Enclosed comments may span multiple lines. Examples:

```
 700 EXCH$ /* TELEPHONE EXCHANGE */ = STR(PHONENUMBER$,4,3)
1100 /* COMMENTS OF THIS FORM MAY EVEN EXTEND
1200  OVER MANY LINES, AND MAY CONTAIN ANY SERIES
1300  OF CHARACTERS... !@#$%¢&*()... BUT MUST
1400  END WITH THE STAR-SLASH SYMBOL:  */
```

2.4.2 Compiler Directives

VS BASIC also lets the programmer use the TITLE and EJECT statements to control the pagination and titling of the program source listing produced by the compiler. TITLE and EJECT both belong to a set of statements known as compiler directives. Lines which contain TITLE and EJECT directives are not printed in source listings generated by the compiler, although their respective effects on the form of the listing do appear, as described below. TITLE and EJECT lines are, however, shown by the VS EDITOR and file display programs.

A TITLE statement must be the only statement on a line. When a TITLE statement is encountered during compilation, the compiler skips to the top of the next page of the output listing and titles that page with the line of text specified in the TITLE statement. All subsequent pages of the listing will also be printed with the specified title until another TITLE statement occurs. All characters (including any occurrence of ":" or "!") following the reserved word TITLE on the same line are regarded as part of the title. Note that this means that a TITLE line cannot be continued by use of the "!" convention described in Subsection 2.3.3, Continuation of Statements. For example, to print the title PART I: VARIABLE INITIALIZATION SECTION at the top of a page of source listing, one would use

```
500 TITLE PART I:  VARIABLE INITIALIZATION SECTION
```

The EJECT statement, which must also appear as the only statement on a line, causes the compiler to skip to the top of the next page of the source listing and to print the most recently specified title at the beginning of the page. All text following the word EJECT on the same line is ignored.

CHAPTER 3
DATA FORMATS

3.1 INTRODUCTION

Programs written in VS BASIC are capable of processing both numeric and alphanumeric data. Numeric data can be stored and processed either in integer format or in floating-point format. Alphanumeric information can be stored and manipulated as single characters or as strings of characters. In addition, individual bits within alphanumeric data can be manipulated using logical operators.

Both numeric and alphanumeric data can be processed singly, as constants or scalar variables, or in sets of arbitrary size called arrays, which can be referred to by a single name. Individual elements of an array can also be processed as scalar variables.

This chapter describes the types of data VS BASIC processes and the formats used for representing data. The various operations which can be performed on data are discussed in Chapters 4 and 5.

3.2 CONSTANTS, VARIABLES, RECEIVERS, AND EXPRESSIONS

A constant is an item of data whose value is fixed in a program and does not change during program execution. In contrast, a variable is an item of data that does not have a fixed value and can be assigned different values during program execution. A constant appears in a VS BASIC program as a number or a literal (see Subsection 3.4.1, Literals (Alphanumeric Constants)). Each variable is represented by a unique variable name that is used to name that area in storage which holds the value of the variable. For example, in the statement

```
CIRCUMF = 3.14159 * DIAM
```

CIRCUMF and DIAM are variable names, and 3.14159 is a constant. This particular statement multiplies the value of the variable DIAM by the constant 3.14159 (the asterisk (*) is the symbol used to indicate multiplication in BASIC) and stores the product in the variable called CIRCUMF. The different types of constants and variables VS BASIC recognizes and the rules for naming variables are described in Subsection 3.3.3, Numeric Variables.

A receiver is a variable into which data can be stored. Receivers are used wherever a value is "received," e.g., on the left side of a LET statement, in the argument list of a READ statement, etc. All variables are receivers; for numeric data, all receivers are variables. Alphanumeric receivers (or simply "alpha receivers") include alphanumeric variables and a few special functions. See Subsection 5.4.2, Alpha Receivers, for a list of all alpha receivers.

An expression is either a constant, a variable, a function, or some combination connected by operators. When a statement containing an expression is executed, the indicated operations and functions are performed to yield a single value for the expression. Functions and operators are constructs which specify particular operations to be performed on one or more expressions. Separate operators and functions exist for manipulating numeric and alphanumeric data, and are discussed in Chapters 4 and 5. An expression can contain either numeric or alphanumeric data, but the two data types cannot be combined in one expression.

3.3 NUMERIC DATA

VS BASIC recognizes two types of numeric data: floating-point and integer. The types are clearly distinguished in BASIC syntax, require different amounts of internal storage, are represented differently in internal format, and have a different range of allowable values.

Integer data, which is used to represent "whole" (i.e., non-fractional) numbers, are stored in four bytes of memory. Floating-point data is stored in eight bytes of memory in the form of: (1) a hexadecimal fraction between 0 and 1, and (2) a power of 16. Integer representation has the advantage that integer operations are considerably faster than floating-point operations. Floating-point representation, on the other hand, provides a convenient way of processing numbers which have either extremely large or extremely small magnitudes, and of operating upon such numbers with a high degree of precision. It is also the only way to store fractional numbers. VS BASIC allows complete freedom to mix both types of data in arithmetic expressions and assignment statements. Expressions containing both integer and floating-point data are called mixed mode and are discussed in Section 4.5, Mixed Mode Arithmetic.

3.3.1 Floating-point Constants

A floating-point constant may be a positive or negative number of up to 15 digits. The compiler will issue a warning when it encounters a floating-point constant with more than 15 digits in the source program. Only the first 15 digits, excluding leading zeros, are used by VS BASIC statements or functions.

NOTE:

All VS BASIC numeric statements and functions use only the first 15 digits of a floating-point number. Specifically, truncation to 15 digits occurs (at the indicated time) with:

1. Any floating-point constant (compile time).

2. Any user-input floating-point value (i.e., to INPUT, ACCEPT, or READ statements; run time).

3. Any floating-point number converted from alphanumeric format to floating-point format via Image (%), FMT, or CONVERT statements (run time).

The magnitude of a floating-point constant can range from zero or approximately 5.4×10^{-79} to 7.2×10^{75}.

Very large or very small floating-point numbers can be expressed in exponential form. Exponential form corresponds to standard "scientific notation" in which numbers are written as a decimal with one digit to the left of the decimal point, multiplied by some power of 10. Since the superscripts needed to write numbers in such notation cannot be easily represented on a keyboard device, a number in exponential form is represented as a decimal (usually with one digit to the left of the decimal point), immediately followed by the letter E, followed by an exponent representing a power of 10. The exponent must be an integer and may have an optional sign; if no sign is given for the exponent, it is assumed to be positive. Leading zeroes may be omitted. Numbers in exponential form contain no embedded spaces between the decimal, the letter E, and the exponent. For example:

Long form Exponential form	Scientific notation	Floating point constant
45000000	4.5×10^{7}	4.5E07
.00000045	4.5×10^{-7}	4.5E-7
37234.123	3.7234123×10^{4}	3.7234123E+04

The following are examples of valid floating-point constants in BASIC:

4, -10, 1432443, -7865, 24.4563, -3E2, 2.6E-27

The following are examples of invalid floating-point constants in BASIC:

<u>8.7E5.8</u> -- Not valid because of the decimal point in the exponent.

<u>.87E-99</u> -- Not valid because it is less than 5.4E-79.

<u>103.2E99</u> -- Not valid because it is greater than 7.2E75.

3.3.2 Integer Constants

An integer constant may range from -2,147,483,648 to 2,147,483,647 (the decimal equivalent of the range of binary numbers which can be represented with 32 bits) and must, as its name indicates, be an integer. An integer constant is denoted by a "%" following the constant. Thus, "4%" is an integer, and "4" is a floating-point number. The percent sign for numeric constants is only permitted for numbers or variables actually contained in the source file. Therefore, numbers given to the program during execution (i.e., from the workstation or data file, or converted from an alpha expression) must be given in floating-point form (i.e., without the percent sign).

3.3.3 Numeric Variables

Numeric variables are used to reference numeric data stored in memory. Unlike constants, variables can be assigned new values during execution by a variety of different statements. Each variable name in a program is associated with an area in memory used to contain the value of that variable. Numeric variables are initialized to zero by the compiler.

As is the case with constant data values, VS BASIC processes scalar variable values as either integers or floating-point numbers. All scalar floating-point variables are eight bytes in length, while all scalar integer variables are four bytes in length.

Within the floating-point and integer data types, VS BASIC variable numeric data can be referred to as either <u>scalar</u> variables or <u>array</u> variables. The two kinds of variables differ in the syntax rules which apply to them and in their storage requirements. A numeric scalar variable contains a single numeric value. An array variable, on the other hand, contains one or more values, or "elements," all of which can be referenced by a single name and which can be manipulated either collectively or individually. Array variables are discussed more fully in Section 3.5, Array Variables.

It is important to note the differences between integer and floating-point calculations. Integer calculations are inherently precise and consistent, while floating-point calculations are approximations and are somewhat inconsistent. The differences between integer and floating-point calculations are described in detail in Appendix D.

Each variable in a program is referred to by an arbitrary and unique variable name chosen by the programmer. A variable name may be any string of up to 64 letters, digits, and underscores, provided that the first character is a letter and that the string is not a VS BASIC reserved word (see Section 2.2, Statements and Appendix A). Numeric variables are designated as integer data type (see previous section) by appending a percent sign (%) to the end of the variable name. Any numeric variable which does not have "%" as the last character of its name is treated as a floating-point variable. The following are examples of valid numeric variable names:

Floating-point	Integer
N	N%
CAT	MOUSE%
PART_2	FIRST_3_LINES%

The following are examples of incorrect variable names:

Floating-point	Integer
2ND_PART	First character must be a letter.
LINE COUNT%	Names cannot contain spaces. COUNT% alone is a legal variable name.
LAST_%ILE	"%" is legal only at the end of a variable name.

Note that a floating-point variable name and an integer variable name always identify different variables, even if the names exclusive of the "%" (i.e., the letters, digits, and underscores) are identical. For example, INFUNDIBULUM and INFUNDIBULUM% identify two different variables, one floating-point and one integer, and both may be used to refer to different items of data in the same program without ambiguity.

3.3.4 Floating Point and Integer Calculation

The type of data representation (floating point or integer) chosen may have a significant effect on the speed and accuracy of a program. Integer calculations are precise and consistent; however, they are somewhat slower than floating-point calculations. Floating-point calculations increase operating speed at the expense of accuracy.

Integer calculations are exact. The standard set of mathematical laws and operations, including the associativity and commutivity of operations and the equality and relational operations produce the expected results using integer calculations.

Floating-point values are approximations. There may be exceptions to the normally expected mathematical results when using floating point values. These floating-point approximations are faster than integer representations but calculations using this data representation are sensitive to the order of operations and the source of the input values.

See Appendix D, Floating Point and Integer Calculations, for a detailed discussion of the differences between these data types as well as suggested techniques for resolving the resulting difficulties.

3.4 ALPHANUMERIC DATA

In addition to its ability to manipulate and operate upon numeric data, VS BASIC also provides the capability for processing information in the form of alphanumeric character strings. A character string is a sequence of characters treated as a unit. A character string may consist of any sequence of keyboard characters, including letters A - Z, digits 0 - 9, and special symbols. Character strings are represented in a program as literal strings (the alphanumeric equivalents of numeric constants), or as the values of alphanumeric string variables. Characters not found on the keyboard can be represented as hexadecimal ASCII codes. Typical examples of uses of character strings are names, addresses, and report headings.

Note that alphanumeric data cannot be operated upon by numeric functions or operators. A separate set of operators and functions exists for the manipulation of alphanumeric data. VS BASIC also provides functions which convert alphanumeric data to numeric form and vice versa. These are discussed in Section 9.1, Data Conversion Statements.

3.4.1 Literals (Alphanumeric Constants)

The value of an alphanumeric data item which is a fixed constant in a source program is called a literal or a literal string. A literal string can be written either by enclosing the desired sequence of characters in quotation marks or by specifying the hexadecimal ASCII codes of the characters in the literal with the HEX function.

One type of quoted alphanumeric literal string is a sequence of 1 to 255 characters enclosed in double quotation marks ("..."). Any keyboard character except the double quote character may appear in a double-quoted literal. Literal strings can be used to specify messages, headings, or titles to be output to some device (e.g., workstation or printer) by any of several output statements. For example,

```
PRINT "LAST PAGE="; LPG
```

In this case, LAST PAGE= is a quoted literal which would be printed exactly as it appears. LPG is the name of a floating-point variable whose value would be printed following LAST PAGE=.

A second type of quoted literal string is available for specifying lowercase characters. The literal string is entered with uppercase characters enclosed in single quotes ('...'). The single quotes indicate that the uppercase letters are to be treated as lowercase by the system. For example,

```
PRINT "J";'OHN';"D";'OE'
```

```
Output:  John Doe (if device is capable of printing lowercase letters)
             or
         JOHN DOE (if device only prints uppercase letters)
```

Any character is valid in a lowercase literal string except the single-quote character ('). A single quote literal string may contain double quotes, and vice versa.

Literals can also be written using the HEX function. In this form, characters in the string are specified by their hexadecimal ASCII codes (sometimes called "hex codes"). Each printing character (and each of the non-printing workstation control characters called Field Attribute Characters) can be represented by a corresponding ASCII code composed of two hexadecimal digits (0 - 9 and A- F; see Appendix E for a list of the ASCII hex codes). In a HEX literal, the ASCII hex codes are placed in parentheses following the word HEX. For example,

```
PRINT HEX(414243)
```

prints the string ABC, since 41, 42, and 43 are the hex codes for the first three letters of the alphabet. This statement is equivalent to PRINT "ABC". HEX(4120422043) corresponds to the same sequence of letters, with spaces (ASCII code 20) between them. Any legal hexadecimal code may be specified in a HEX literal string. The user should, however, be aware of the special use of hex codes 80 - FF (see Chapter 7, Section 7.3, Subsection 7.3.4, Field Attribute Characters (FACs)).

Literal strings can also be assigned as values to alphanumeric variables. Assignment and other alphanumeric operations are discussed in Section 5.2, Alphanumeric Operators.

3.4.2 Alphanumeric Variables

Alphanumeric character strings can be stored and processed in an alphanumeric string variable (or simply "alpha variable"). Values stored in alpha variables can be stored and processed singly, as scalar variables, or in groups, as array variables. Alphanumeric and numeric arrays are discussed in Section 3.5, Array Variables. The following discussion applies to alphanumeric scalar variables.

Alpha variable names, like those of numeric variables, are sequences of up to 64 letters, digits, and underscores, provided that the first character of the name is a letter and that the name is not a VS BASIC reserved word (see Appendix A). Alpha variable names are distinguished from numeric variable names by a dollar sign ($) appended to the end of the variable name. For example, the variable name THING refers to a floating point numeric variable, whereas THING$ refers to an alphanumeric variable. Similarly, ITEM% is an integer variable; ITEM$ is an alpha variable. A numeric variable and an alpha variable are separate and independent entities, even if they have the same name exclusive of the "$".

An alphanumeric variable identifies a unique location in memory reserved for the storage of alphanumeric data. The compiler reserves space for each variable during compilation, at which time the program is scanned for all variable references. The number of characters which can be stored in an alpha variable depends on how much space is reserved for that variable during compilation. Each character requires one byte (eight bits, or binary digits) of storage. The amount of space reserved for each variable can be specified by the programmer in a DIM or COM statement. For example,

```
DIM WORD$ 10, LINE$ 80
COM HORSE$ 10, COW$ 17
```

reserves 10 bytes of storage for WORD$, 80 bytes for LINE$, 10 bytes for HORSE$, and 17 for COW$, the latter two in the common storage area. (For an explanation of common storage, see Subsection 6.5.4, Passing Values to External Subroutines.) An alpha scalar variable may be specified as being of any length between one and 256 characters (bytes). An alpha variable may not appear more than once in DIM or COM statements in a program. If the programmer does not explicitly dimension an alpha variable in a DIM or COM statement, the compiler automatically reserves 16 bytes for the variable. The DIM and COM statements are both also used for dimensioning arrays (see Subsection 3.5.2, Dimensioning an Array); the COM statement is also used for placing variables of any type in common storage (see Subsection 6.5.4, Passing Values to External Subroutines and the COM statement entry in Part 2).

NOTE:

Any alpha variable which has not had some other value assigned to it is defined as being filled with blanks (ASCII code HEX(20)).

The length of an alpha variable or alpha array element specified in a DIM or COM statement is called its "defined" length. In many cases, however, the character string stored in an alpha variable will not occupy the entire defined length. The last character of an alpha variable is normally taken to be the final nonblank character (except when the value is all blanks, in which case the value is treated as one blank). Hence, trailing blanks generally are not considered part of the value of an alpha variable. For example:

```
100 A$="ABC    "
200 PRINT A$;"DEF"
```

Output: ABCDEF (Note that the trailing blanks of A$ were not printed.)

The character string stored in an alpha variable is called the "current value" of the alpha variable, and its length, up to the first trailing blank, is called the "current length" (or "actual length") of the variable. The length function, LEN, determines the current length of an alpha variable. For example:

```
100 A$="ABCD    "
200 PRINT LEN(A$)
```

Output: 4   (Trailing blanks are not considered to be part of the value of an alpha-variable by LEN.)

Most alphanumeric operators and functions operate on the current value of an alpha variable. In some cases (e.g., ACCEPT and DISPLAY statements), however, the entire defined length of the variable may be used. It is therefore important to understand the distinction between defined length and current length.

NOTE:

If the defined length of an alpha variable is greater than necessary for storing the value of a given alpha expression, the variable is padded with blanks (ASCII code HEX(20)) when the value is assigned.

3.5 ARRAY VARIABLES

An "array variable" is a collection of scalar variables identified by a common name. Each scalar variable contained in the array is called an "element" of the array, and can be identified by specifying the array name followed by a subscript or pair of subscripts, which locate the element within the array. Arrays, like scalar variables, may hold floating point, integer, or alphanumeric data. A single array cannot hold values of more than one type. The names of array variables are formed in the same way as the names of scalar variables (a sequence of 1 to 64 letters, digits, and underscores, as described in Subsection 3.3.3, Numeric Variables; names of integer arrays must end in '%' and those of alpha arrays must end in '$'). The one additional restriction on array names is that they cannot begin with the characters FN.

NOTE:

Any attempt to use a name beginning with FN for an array will either result in an error message at compilation time or in a logically incorrect object program. Any name beginning with FN and containing parentheses is interpreted by the compiler to refer to a user-defined function (see Subsection 4.4.2, User-Defined Functions).

In general, any reference to an array variable must consist of the array name followed by parentheses. If the parentheses enclose an expression or a pair of expressions, the expressions are interpreted as the subscripts of a particular element in the array. For example, the fifth element in floating-point array N() could be specified as N(5); BOX$(K) refers to the K-th element of the alpha array BOX$(). Note that the subscript is enclosed in parentheses immediately following the array name. In situations in which the entire array (rather than a particular element of the array) is to be referenced, the array name must be followed by empty parentheses (e.g., N() or BOX$()) to form an "array-designator." The array name alone (e.g., N or BOX$) is used only in special matrix statements (e.g., MAT INPUT and MAT PRINT).

Since scalar variables are different from array variables, the same name (i.e., the same sequence of letters, digits, and underscores) may be used both as a scalar variable name and as an array variable name. Thus N() designates an array variable, while N names a scalar variable, except in a matrix statement. Except in matrix statements (see Section 9.2, Matrix Statements), the array must always be referenced with an array-designator to indicate an array rather than a scalar variable. For example:

```
WHALE -- identifies a floating-point scalar variable.
WHALE% -- identifies an integer scalar variable.
WHALE() -- identifies a floating-point array.
WHALE%() -- identifies an integer array.
WHALE$ -- identifies an alphanumeric scalar variable.
WHALE$() -- identifies an alphanumeric array.
```

To minimiz the chance of confusion however, use of the same name for scalar and array variables in a program is not recommended.

3.5.1 One-Dimensional and Two-Dimensional Arrays

Array variables either one-dimensional or two-dimensional. A one-dimensional array is a list of all variables identified by the same name. A two-dimensional array is a table of variables all identified by the same name.

A one-dimensional array can be conceived of as a list or column of variables (elements), each occupying its own slot, or row, in the column. Consider, for example, the representation of array DWARF() in Figure 3-1.

	DWARF()
Row 1	DWARF(1)
Row 2	DWARF(2)
Row 3	DWARF(3)
Row 4	DWARF(4)
Row 5	DWARF(5)

Figure 3-1. The One-Dimensional Array DWARF()

Note that DWARF() contains a total of five elements and that each element is identified by specifying its row. For example, element DWARF(3) is located in Row 3.

One-dimensional arrays are also called "lists," "vectors," "column vectors," and, since each element is identified by a single subscript, "singly-subscripted arrays."

The scheme in Figure 3-1 can be generalized to contain two or more columns. When this is done, the result is a two-dimensional array. A two-dimensional array can be conceived of as a table consisting of two or more columns of elements. Consider, for example, the representation of the two-dimensional array HOBBIT() in Figure 3-2.

		HOBBIT()	
	Column 1	Column 2	Column 3
Row 1	HOBBIT(1,1)	HOBBIT(1,2)	HOBBIT(1,3)
Row 2	HOBBIT(2,1)	HOBBIT(2,2)	HOBBIT(2,3)
Row 3	HOBBIT(3,1)	HOBBIT(3,2)	HOBBIT(3,3)
Row 4	HOBBIT(4,1)	HOBBIT(4,2)	HOBBIT(4,3)
Row 5	HOBBIT(5,1)	HOBBIT(5,2)	HOBBIT(5,3)

Figure 3-2. The Three-Dimensional Array HOBBIT()

Note that HOBBIT() consists of three columns of elements, with five rows in each column, for a total of 15 elements. In this case, it is not sufficient to identify each element by its row, since the element may be in Column 1, Column 2, or Column 3. A second subscript is required to identify the column. The convention followed when referencing a particular element in a two-dimensional array is always to specify the row first, and then the column. Thus HOBBIT(3,2) identifies the element in Row 3 and Column 2.

Two-dimensional arrays are also called "tables," or "matrices," and, because each element is identified by a pair of subscripts, "doubly-subscripted arrays."

Elements in an array can be referred to by subscripts that are legal BASIC expressions. Thus JIM(N) refers to the N-th element of array JIM() for whatever value N has at the time of execution. This ability to reference an array by a variable subscript is one of the useful features of arrays, since it can eliminate a considerable amount of repetitive coding.

For example, the following three statements

```
100  FOR I = 1 TO 50
200  PRINT JIM(I)
300  NEXT I
```

will cause the first 50 elements of array JIM() to be printed with considerably less coding than 50 consecutive PRINT statements.

> NOTE:
>
> If the value of an expression used as a subscript is not an integer at run time, the value of the expression is truncated and the integer value is used as the subscript.

3.5.2 Dimensioning an Array

When a program is compiled, the BASIC compiler reserves storage space for each variable. To do this, the compiler must know how much space to allocate for each variable. Since arrays may be either one- or two-dimensional and may contain varying amounts of data, the programmer must tell the compiler how much space to reserve for each array in a program; the array must be dimensioned. An array is dimensioned by specifying whether it has one or two dimensions and how many rows (and columns, if two-dimensional) are in the array. Dimension information is specified using either the DIM (dimension) or COM (common) statement. For example, to allocate space for a one-dimensional integer array of 10 elements named VEGETABLE%(), one would write

```
DIM VEGETABLE%(10)
```

If VEGETABLE%() is to be used by more than one program or subprogram running together, one would use COM instead of DIM .

DIM and COM statements may be used to define any number of arrays of any type, as long as each array is separated from the one following it by a comma. When using DIM or COM to dimension an alpha array, the length of each element in the array can be specified as an integer immediately following the right parenthesis.

For example,

```
DIM NAME$(500)10, CITY(100), STATE(5,10)
COM CODE$(20,10)5, ZIP%(1000), COUNT%
```

defines a 500-element one-dimensional alphanumeric array (NAME$()) where each element is 10 bytes long, a 100-element one-dimensional floating-point array (CITY()), a 5-row by 10-column two-dimensional floating point array (STATE()), a 20-row by 10-column two-dimensional alpha array (CODE$()) with each element 5 bytes long, and a 1000-element one-dimensional integer array (ZIP%). The latter two arrays are designated as common, as is the integer scalar COUNT%. The use of DIM and COM statements to specify the length of alpha scalars is discussed in Subsection 3.4.2, Alphanumeric Variables.

If an array is not dimensioned before its first occurence in an executable program statement, the compiler automatically assigns default dimensions of 10 rows by 10 columns. In the case of alpha arrays, each element is assigned a default length of 16 bytes. Therefore, any array which is to be of any other dimension must be dimensioned <u>before</u> its first occurence in an executable statement. No array may be dimensioned more than once in a program. Row and column dimensions specified in DIM or COM statements must be between 1 and 32,767.

The total size of all the variables in a program, including array variables, is limited to a maximum of not more than 512K (524,288) bytes. If the variables in a source program require more than 512K bytes of storage, the compiler outputs an error message and halts code generation.

Although programs may be compiled with up to 512K bytes of variable storage space (Segment 2 space), the resulting object program cannot be executed unless there is sufficient space available on the particular VS configuration at run time. In other words, the fact that a particular program compiled successfully on a particular system does not guarantee that it will also run on that system. For example, on a VS system with only 256K bytes of Segment 2 space allocated to each task, a program requiring 400K of Segment 2 space would compile with no errors (assuming it were syntactically correct), but would not run due to insufficient memory to store the variables during execution.

Since DIM statements are processed during compilation, prior to program execution, they <u>cannot</u> be supplied with variable subscripts, since the value of the variable is unknown at that time. The following statement, for example, produces an error message:

```
DIM A1(5,N)
```

CHAPTER 4
NUMERIC OPERATIONS

4.1 INTRODUCTION

Numeric data (see Section 3.3, Numeric Data) is manipulated in VS BASIC by means of operators and functions. An operator is a symbol (such as + or -) which specifies some operation (such as addition or subtraction) to be performed, usually involving two numeric quantities. A function is a construct that performs some series of operations on one or more input values (called arguments) and returns a single output value. For example, SIN(X) and SQR(X) are functions which calculate the sine and square root of an argument, in this case of the variable X. A number of numeric constants, variables, and functions connected by numeric operators constitutes a numeric expression. When values are supplied for any variables in an expression, the value of the expression is determined by performing the indicated operations and functions. This occurs when a statement containing an expression is executed when a program is run. The value of the expression is then used in whatever way is indicated by the particular statement being processed.

4.2 NUMERIC OPERATORS

There are three types of numeric operators used in BASIC: assignment, arithmetic, and relational. The assignment operator assigns a value to a particular variable. The arithmetic operators specify the basic arithmetic operations which can be performed on numeric quantities: addition, subtraction, multiplication, division, exponentiation, and negation. Relational operators specify comparisons to be made between two numeric values so that a program may take different actions depending on whether one value is greater than, equal to, or less than another.

4.2.1 The Assignment Operator

The equals sign (=) is the assignment operator used only in assignment statements. An assignment statement stores the value of the expression on the right of the equal sign in the variable(s) named to the left of the equal sign. An assignment statement consists of the optional reserved word LET, followed by one or more variable names, followed by the equal sign, followed by a numeric expression. For example,

```
LET SUM=A+B
LET SQUARE(5,17)=(ZONK-POW)+10
LET SLITHEY_TOVES, MOME_RATHS = VORPAL/FRUMIOUS
```

The keyword LET may be omitted:

```
DIFFERENCE=A-B
CABBAGES, KINGS=10
```

Note that the equal sign has a different meaning in contexts other than assignment statements (see Subsection 4.2.3, Relational Operators).

4.2.2 Arithmetic Operators

The following symbols are used as arithmetic operators.

Symbol	Operation	Sample Expression	Explanation
↑ or **	exponentiation	A↑B or A**B	Raise A to the power B.
*	multiplication	A*B	Multiply A by B.
/	division	A/B	Divide A by B.
+	addition	A+B	Add B to A.
-	subtraction	A-B	Subtract B from A.
-	unary negation	-A	Negate A.

NOTE:

All arithmetic operations must be explicitly specified. While in normal algebraic notation, expressions such as AB or A(B) may be used to indicate multiplication; the operation must be explicitly specified (e.g., A*B.)

When a numeric expression is evaluated, arithmetic operations are performed in the following order or hierarchy:

1. All operations within parentheses are performed. The innermost parenthesized expressions are evaluated first.

2. All unary negation (-) and exponentiation (↑ or **) operations are performed (left to right).

3. All multiplication (*) and division (/) operations are performed (left to right).

4. All addition (+) and subtraction (-) operations are performed (left to right).

NOTE:

Every arithmetic operator must be followed by a numeric expression. Thus it is not permissible to have an operator immediately followed by another operator, as in A-*B. To indicate an operation on the negative of an expression, parentheses must be used to enclose the expression and the negating minus sign. For example, A*(-B) is permissible, but A*-B is not.

When there are no parentheses in the expression and the operators are at the same level in the hierarchy, the expression is evaluated from left to right. Parentheses may be used to group operations and so alter the order of evaluation of terms within an expression. Quantities within parentheses are evaluated before the parenthesized quantity is used in further computations. For example:

A*B/C	A is multiplied by B; the product is then divided by C.
A*(B/C)	B is divided by C; the quotient is then multiplied by A.
X+Y*Z	Y is multiplied by Z (multiplication precedes addition by Rule 3 above); the product is then added to X.
(X+Y)*Z	Y is added to X; the sum is then multiplied by Z.

Parentheses may be "nested" to any level. That is, a parenthesized expression may contain other parenthesized expressions, as in A+(B-((C/D)**2)). In such cases, the expression within the innermost set of parentheses is evaluated first, and evaluation proceeds to the outermost set of parentheses. For every left parenthesis there must be one matching right parenthesis at some later point in the expression.

When in doubt, parentheses may always be used to insure that a complex expression is evaluated in the intended way. Redundant parentheses have no effect on the order of evaluation of an expression.

4.2.3 Relational Operators

Relational operators are used in IF...THEN statements when values are to be compared.

For example, when the statement

```
IF G<10 THEN 60
```

is executed, if the value of G is less than 10, processing continues at program line number 60. Otherwise, execution continues in the normal sequence with the statement following the IF statement.

The following relational symbols are used in VS BASIC:

Symbol	Sample Relation	Explanation
=	A = B	A is equal to B.
<	A < B	A is less than B.
<=	A <= B	A is less than or equal to B.
>	A > B	A is greater than B.
>=	A >= B	A is greater than or equal to B.
<>	A <> B	A is not equal to B.

These symbols are also used in the POS function and the SEARCH statement (see Section 5.6, Numeric Functions with Alpha Arguments, and the SEARCH statement entry in Part 2).

4.3 NUMERIC EXPRESSIONS

A numeric expression is either one or a series of constants, variables, or functions, connected by arithmetic operators. Numeric expressions can be evaluated in a variety of different BASIC statements. In the following examples valid numeric expressions are boxed:

PHONE = $\boxed{\texttt{2988}}$

INDEX = $\boxed{\texttt{(VARX-OFFSET)/LOG(T↑2)}}$

PRINT $\boxed{\texttt{SIN(THETA)}}$

FOR I = $\boxed{\texttt{3+K2}}$ TO $\boxed{\texttt{4*Y}}$ STEP $\boxed{\texttt{D(3+K)-1}}$

Most commonly, expressions are evaluated and their values assigned to variables in assignment (LET) statements (see Subsection 4.2.1, The Assignment Operator), or they are evaluated and their values printed or displayed in PRINT statements. Operations in an expression are performed in sequence from highest priority level to lowest (see Subsection 4.2.2, Arithmetic Operators).

4.4 NUMERIC FUNCTIONS

A numeric function is a construct in the BASIC language which takes one or more numeric expressions as input values (called arguments), performs some series of operations on them, and returns a single numeric output value. The value of the function may be used anywhere a numeric expression is allowed, such as on the right-hand side of an assignment statement or as part of a larger numeric expression.

Syntactically, a function is written as follows:

```
fname[(argument[,argument][,...])]
```

where "fname" is the name of a function, and "argument" is any expression acceptable to the particular function used. Expressions used as the arguments of a function are evaluated before the computation indicated by the function is performed. The result of this computation may be used as part of a larger expression. For example,

```
100 LET X=SIN(Y/2)+1
```

causes: (1) the expression Y/2 to be calculated, (2) the sine of that expression to be determined (SIN is the function name), (3) 1 to be added to the sine, and (4) the assignment of the final value to the variable X.

VS BASIC recognizes two major kinds of numeric functions: intrinsic and user-defined functions.

4.4.1 Intrinsic Functions

Intrinsic (or "built-in") functions are defined within the BASIC language and may be used at any time in a program. The twenty-five intrinsic functions recognized by VS BASIC include all mathematical functions such as trigonometric, absolute value, and logarithmic functions. A random number generator and various functions specialized for use in data processing are also available. The intrinsic numeric functions vary in the type and number of arguments that they require, but all return numeric values.

In addition to the intrinsic functions, VS BASIC also has an intrinsic named constant, which is PI. PI may be used anywhere a numeric expression is allowed, and has the value 3.14159265358979.

The intrinsic numeric functions are discussed below and in Part 2 of this manual.

The SIZE function, which deals with file I/O, is discussed in Section 8.5, Intrinsic File I/O Functions.

Trigonometric Functions

The sine, cosine, tangent, arcsine, arccosine, and arctangent functions are available in BASIC. Other trigonometric functions can be easily expressed using combinations of these functions. Each of these functions takes one numeric argument, and returns a floating point value.

Function	Meaning
SIN(x)	the sine of x.
COS(x)	the cosine of x.
TAN(x)	the tangent of x.
ARCSIN(x)	the inverse sine (Arcsine) of x.
ARCCOS(x)	the inverse cosine (Arccosine) of x.
ARCTAN(x)	the inverse tangent (Arctangent) of x.
ATN(x)	same as ARCTAN (ATN is a synonym for ARCTAN).

These functions can express and accept angular measure in degrees, radians, or grads (400 grads = 360 degrees). Radian measure is used as the default in every program or subroutine, until one of the following statements is encountered:

SELECT DEGREES -- which selects degrees.
SELECT GRADS -- which selects grads.
SELECT RADIANS -- which selects radians.

The mode used at any time is determined by the most recently executed SELECT statement in that program or subroutine. For instance, a program can execute a SELECT DEGREE statement, thus changing the trig mode to degrees. If it then uses the CALL statement to call a subroutine, the mode becomes radians, assuming the subroutine has not previously reset the mode. If the subroutine executes a SELECT GRADS statement, the mode for subsequent trigonometric functions becomes grads. When the END statement is executed, returning control to the calling program, the mode reverts to degrees. If that subroutine is called again, the initial mode will be grads.

The SELECT statement is discussed further in Part 2. The arguments of the sine, cosine, and tangent functions will be interpreted as degrees, grads, or radians depending on the SELECT setting in effect at the time of execution. The values returned by the inverse trigonometric (arc) functions are likewise in degrees, grads, or radians according to the SELECT setting.

Other Numeric Functions

The remaining eighteen numeric functions are described below. For more detailed descriptions of these functions (including which functions return integer values and which return floating point values) see the appropriate entries in Part 2.

Function	Meaning	Number and type of arguments
ABS(x)	The absolute value of the argument: -x if x < 0; x if x >= 0.	1 numeric.
DIM(x(),d)	The maximum 1st or 2nd subscript of the array x.	1 array designator (x()), 1 integer (d) = 1 or 2.
EXP(x)	The exponential function; "e" (2.718...) raised to the x-th power.	1 numeric.

Function	Meaning	Number and type of arguments
INT(x)	The greatest integer less than or equal to x.	1 numeric.
LEN(a$)	The actual length, in bytes, of a$.	1 alphanumeric.
LGT(x)	Common (base 10) logarithm of x.	1 numeric.
LOG(x)	Natural (base "e") logarithm of x; inverse function of EXP.	1 numeric.
MAX(x,y,z)	The value of the largest element in the argument list.	1 or more numeric scalars or numeric array designators.
MIN(x,y,z)	The value of the smallest element in the argument list.	1 or more numeric scalars or numeric array designators.
MOD(x,y)	The modulus function; the remainder of the division of x by y.	2 numeric.
NUM(a$)	The number of sequential ASCII characters in a$, starting with the first character, that represent a legal BASIC number.	1 alphanumeric.
POS(a$<b$)	The position of the first character of a$ which is < , <=, >, >= , <>, or = the first character of b$.	2 alphanumeric.
RND(x)	A pseudo-random number between zero and one.	1 numeric.
ROUND(x,n)	The value of x, rounded off to n decimal places.	1 numeric (x), 1 integer(n).
SGN(x)	The signum function; -1 if x is negative, 0 if x is zero, or +1 if x is positive.	1 numeric.
SIZE(#n)	The size in bytes of the most recently read record from file #n. (See Section 8.5, Intrinsic File I/O Functions).	1 integer file-expression.

Function	Meaning	Number and type of arguments
SQR(x)	The square root of x.	1 numeric.
VAL(a$,d)	The numeric value of the first d bytes of a$.	1 alphanumeric (a$), 1 integer (d) = 1,2,3, or 4.

The DIM, RND, and ROUND functions are discussed here in more detail.

DIM

The DIM function (not to be confused with the DIM statement) requires two arguments: the first must be an array-designator (the array name plus parentheses, e.g., A()) occurring in the BASIC program; the second must be an expression whose value is either 1 or 2. The DIM function returns either the row or column dimension of the named array.

```
DIM(X(),1) -- returns the row dimension of the array X.
DIM(X(),2) -- returns the column dimension of the array X.
```

RND

The RND (random number) function is used to produce a pseudo-random number between 0 and 1. The term "pseudo-random" is used because a digital computer cannot produce truly random numbers. Instead, each time the RND function is called, it uses an internally-stored number as a "seed" from which to generate the next "random" number by a fixed internal algorithm. Since the algorithm is always the same, it will always produce the same value for a given seed value. By calling the RND function repeatedly and using the output value of each call as the seed for the next one, a sequence of numbers is generated which, though obviously not truly random, is scattered about in the range zero to one in such a manner as to appear random; thus the term "pseudo-random."

There are three different ways in which the RND function can be used: (1) to generate a pseudo-random number based on a seed value; (2) to reset the seed value to some number specified by the user program (as either a constant or a variable); (3) to reset the seed value based upon the time-of-day clock when the program is run. Which mode of operation is selected depends upon the value of the expression used as an argument in the function RND(expn) as follows:

1. expl < 0 or expl $\geq$ 1

   If the argument (expl) is less than zero or greater than or equal to one, RND produces a pseudo-random number from the seed value. If this is the first use of RND in the program, the seed has a value set by the BASIC compiler during compilation. Otherwise, it has the value produced by the last RND call executed.

2. 0 < exp2 < 1

If the argument (exp2) is between zero and one, RND returns the argument itself as the result and resets the seed to this value. The next use of RND, as in Option 1 above, will use the value of the previous argument (exp2) as the seed from which to generate a pseudo-random value. This allows the user to produce the same sequence of random numbers any number of times within the same program or within different programs.

3. exp3 = 0

If the argument is equal to zero, RND produces a number whose value is computed from the time of day when the RND function is executed, rather than from a user or compiler specified value. This option can be used to reset the seed to a random value, so that on subsequent calls using Option 1, a more seemingly random series of numbers will be produced.

Note that although Option 3 produces a random number in the sense that it will generally differ each time this option is used, repeated RND calls using this option within a program will not produce a dependably random list of numbers. (This is because the relation between successive numbers in such a list will be a function of the time elapsed between function calls.) To produce a more random list, use Option 3 once, followed by as many Option 1 calls as desired. Option 3 should be used only to reset the random number list to a new starting value, not to produce such a list.

Example:

```
100 LET A= RND(.5)
200 LET B= RND(2)
300 LET C= RND(2)
400 PRINT "A=";A, "B=";B, "C=";C
```

Result:

```
A=.5   B=.259780899273209   C=.2989807370711264
```

Every time this program is run, it will produce the same list of numbers.

ROUND

ROUND(X,N) is equivalent to the expression:

```
SGN(X)*( INT(ABS(X)*10↑(INT(N))+0.5)/10↑(INT(N)) )
```

Its effect, is to round off the value of X to the precision specified by N. If N is positive, X is rounded off so that the last significant digit of the function value is the N-th digit to the right of the decimal point. If N is negative, X is rounded off so that the last significant digit of the function value is the (1-N)th digit to the left of the decimal point.

For example:

```
ROUND(123.4567,4) = 123.4567
ROUND(123.4567,3) = 123.4570
ROUND(123.4567,2) = 123.4600
ROUND(123.4567,1) = 123.5000
ROUND(123.4567,0) = 123.0000
ROUND(123.4567,-1)= 120.0000
ROUND(123.4567,-2)= 100.0000
ROUND(123.4567,-3)=   0
```

4.4.2 User-defined Functions

User-defined functions enable the programmer to specify any sequence of numeric operations to be performed on a single numeric argument and to identify that sequence of operations by a function name. Functions are defined using the DEF statement. The DEF statement has the form

```
DEF fname[%](arg) = expr
```

where fname is the function name, arg is a "dummy argument," and expr is any valid numeric expression. Function names are formed according to the same rules which apply for naming scalar variables (see Subsection 3.3.3, Numeric Variables). Although it is permissible to have functions with the same names as variables in a program, this is not recommended. The dummy argument "arg" is used simply to indicate the position in the function definition which is to be taken by the argument value when the function is called, and may be any valid variable name. For example,

```
DEF AREA(X) = 3.14159265 * X**2
```

defines a function which determines the area of a circle given the radius. In this case, AREA is the function name, and X is a dummy argument. The function can be called by a statement elsewhere in the same program, such as

```
LET SEMICIRC = AREA(RADIUS)/2
```

When this statement is executed, the expression in the DEF statement is evaluated, with the value of RADIUS substituted for X. This value is returned to the LET statement, and is then divided by 2. The resulting value is assigned to the variable SEMICIRC.

NOTE:

A function may be defined anywhere in a program, but if the first use of a function precedes its definition, the function name must begin with the characters FN. Otherwise, the BASIC compiler will interpret the function call as an array name reference. This will result in either an error message at compilation time or in logic errors in the program.

4.5 MIXED MODE ARITHMETIC

BASIC allows mixed-mode arithmetic, i.e., floating-point or integer variables may be assigned either floating-point or integer values, with floating-point values truncated to integers. Specifically,

1. Assignment ([LET]) statements allow mixed-mode assignment.

2. Statements performing implicit assignment, such as CONVERT, GOSUB'(), INPUT, ACCEPT, READ, and calls to user-defined functions allow mixed-mode. The only exceptions to this are the CALL and SUB statements, which do not allow mixed mode argument passing.

3. The percent sign (%), used to indicate an integer value, may only be used as a numeric symbol when it appears as such in the source file; in particular, INPUT, ACCEPT, GET, READ, and CONVERT do not allow % as numeric input. Thus, floating-point constants must be used.

4. Expressions in integer syntax are also treated like mixed-mode assignments (truncated to integer) (e.g., BIN(expr), END expr, ON expr GOTO..., RESTORE expr).

4.6 SUMMARY OF NUMERIC DATA TYPES AND TERMS

4.6.1 Floating-Point Data

The allowable range of magnitudes for floating-point values is from approximately 5.4×10^{-79} to 7.2×10^{75}. Floating-point values with magnitudes outside of this range cause conditions called underflow (magnitude too small) and overflow (magnitude too large).

The following are classified as floating-point values:

1. The value of any floating-point variable (no "%" or "$").

   Examples: A, OPHIDIAN, FRUIT(3), D4(X,5)

2. Any numeric constant with no "%."

   Examples: 5, 3.7, -6.321E3, 1E-1

3. The result of any valid numeric function except LEN, NUM, POS, VAL, SGN, SIZE, DIM, ABS(integer), user-defined integer functions (function name ends in "%") and under certain conditions MIN, MAX, and MOD.

   Examples FN2(2%), SIN(3), ABS(-12)

4. The result of any binary operation (+, -, *, /, ↑, **) or MOD function whose two arguments are <u>not both integers</u>.

   Examples: 2/5, 3%↑17, 4%+SQR(16)

5. The result of the MAX and MIN functions when the arguments are not all integers.

> <u>NOTE</u>:
>
> If the evaluation of any numeric expression during program execution results in a floating-point value with magnitude greater than approximately 7.2 x 10^{75} (i.e., an overflow condition), an error occurs, program execution halts, and an appropriate error message is displayed on the workstation. Evaluation of expressions with magnitudes less than 5.4 x 10 (underflow) are treated as zero and do not cause an error. A numeric <u>constant</u> which is either too large or too small causes an error during compilation.

4.6.2 <u>Integer Data</u>

The allowable range of values for integer values is from -2,147,483,648 to 2,147,483,647.

The following are classified as integer values.

1. The value of any integer variable (variable name ending with "%").

   Examples: A%, OPHIDIAN%, FRUIT%(3), D4%(X,5)

2. Any integer constant, which must contain a trailing "%," no decimal point, and no exponent.

   Examples: 375%, -10000%, 2%

3. The result of the numeric functions LEN, NUM, POS, VAL, SGN, SIZE, DIM, ABS (integer), and user-defined integer functions (function names ending in "%").

   Examples: FN3%(7.5), SGN(THE_TIMES), SIZE(#5)

4. The result of any binary operation (+, -, *, /, ↑, **) or MOD function whose two arguments are both integers.

   Examples: 2%/5%, 3%↑(17%), 4% + LEN(B$)

5. The result of the MAX and MIN functions when the arguments are all integers.

> **NOTE:**
>
> If the evaluation of any numeric expression during program execution results in an integer value outside the allowable range (-2,147,483,648 to 2,147,483,647), an error occurs, program execution halts, and an appropriate message is displayed on the workstation.

4.6.3 Numeric Terms

1. Constant: $\left[\begin{Bmatrix}+\\-\end{Bmatrix}\right] \begin{Bmatrix}\text{floating-point constant}\\\text{integer constant}\end{Bmatrix}$

2. Expression †: (or exp)

$$\begin{Bmatrix}\text{numeric variable}\\ \\ \text{constant}\\\text{mathematical function}\\\text{DIM function}\\\text{LEN function}\\\text{NUM function}\\\text{POS function}\\\text{SIZE function}\\\text{user-DEFined function}\\\text{VAL function}\\ \left[\begin{Bmatrix}+\\-\end{Bmatrix}\right]\text{ expression }\left[\begin{Bmatrix}+\\-\\*\\/\\\uparrow\\**\end{Bmatrix}\text{ expression}\right]\ldots\\ \text{(expression)}\end{Bmatrix}$$

3. Integer (or int): digit [digit]...%

† Adjacent operators are not allowed (e.g., A++B).

4. <u>Numeric Array-designator</u>: letter $\left[\begin{Bmatrix}\text{letter}\\\text{digit}\\\text{underscore}\end{Bmatrix}\right]$ ... [%] ()

(not to exceed 64 letters, digits, and/or underscores)

5. <u>Numeric Array Name</u>: letter $\left[\begin{Bmatrix}\text{letter}\\\text{digit}\\\text{underscore}\end{Bmatrix}\right]$ ... [%]

(not to exceed 64 letters, digits, and/or underscores)

6. <u>Numeric Array Variable</u>: letter $\left[\begin{Bmatrix}\text{letter}\\\text{digit}\\\text{[,exp])}\\\text{underscore}\end{Bmatrix}\right]$ ... [%]

(not to exceed 64 letters, digits, and/or underscores)

7. <u>Numeric Scalar Variable</u>: letter $\left[\begin{Bmatrix}\text{letter}\\\text{digit}\\\text{underscore}\end{Bmatrix}\right]$ ... [%]

(not to exceed 64 letters, digits, and/or underscores)

8. <u>Numeric Variable</u>: $\begin{Bmatrix}\text{numeric scalar variable}\\\text{numeric array variable}\end{Bmatrix}$

9. <u>Mathematical Function</u>: PI, ABS, ARCCOS, ARCSIN, ARCTAN, ATN, COS, EXP, INT, LGT, LOG, MAX, MIN, MOD, RND, ROUND, SGN, SIN, SQR, TAN functions.

CHAPTER 5
ALPHANUMERIC OPERATIONS

5.1 INTRODUCTION

Alphanumeric data (or simply "alpha data") is manipulated in VS BASIC by alphanumeric operators and functions. Alpha operators and functions are different from their numeric counterparts described in the previous chapter, even though in some cases (such as assignment statements) the same symbol may be used as either a numeric or an alphanumeric operator. In these cases, the meaning of the symbol is inferred from the data type of the two operands involved. Functions which return alpha values are called alpha functions. In addition, there are some numeric functions which take alpha arguments. Both are discussed below. A series of literals, alpha variables, or alpha functions connected by alpha operators constitutes an <u>alpha expression</u>.

In addition to facilities for manipulating characters and strings of characters, VS BASIC also provides <u>logical functions</u> which enable the programmer to specify operations on individual bits within a stored character. It is also possible to convert alphanumeric representations of numbers to numeric form and vice versa using the CONVERT statement, which is discussed in Part 2 of this manual.

5.2 ALPHANUMERIC OPERATORS

The alphanumeric operators in VS BASIC are the assignment operator, the concatenation operator, and relational operators.

5.2.1 The Assignment Operator

The equal sign (=) is used as the alphanumeric assignment operator in a LET statement, just as it is for numeric assignment. An alphanumeric assignment statement consists of the reserved word LET (optional), followed by one or more alphanumeric variable names or other alpha receivers (see Subsection 5.4.2, Alpha Receivers) followed by the equal sign, followed by an alpha expression. When the statement is executed, the value of the alpha expression is stored into the variable(s) or receiver(s) one character at a time, left to right. This continues until all the characters of the evaluated expression are used up or until the alpha variable or receiver is full. Thus, if the defined length of the alpha receiver is less than the length of the evaluated expression, the rightmost characters of the value of the expression are lost. If the defined length of the alpha variable or receiver is greater than the length of the value of the alpha expression, the remaining bytes of the alpha expression are filled with trailing blanks.

For example:

```
    100 DIM A$50, B$5, C$10, D$10
    200 A$="THE TIME HAS COME, THE WALRUS SAID"
    300 B$=A$
    400 C$=B$
    500 D$=A$
    600 PRINT A$ : PRINT B$ : PRINT C$ : PRINT D$
```

Output:

```
    THE TIME HAS COME, THE WALRUS SAID
    THE T
    THE T
    THE TIME H
```

5.2.2 The Concatenation Operator

The concatenation operator (&) combines two strings, one directly after the other, without intervening characters. The two strings combined by the concatenation operator are treated as a single string.

```
    100 A$="WASTE"
    200 B$="LAND."
    300 C$=A$ & B$
    400 PRINT C$
```

Output:

```
    WASTELAND.
```

Literal strings expressed as constants can be concatenated with literal strings stored as the values of alpha variables. For example:

```
    100 A$="BY"
    200 B$="T.S. ELIOT"
    300 C$=A$ & " " & B$
    400 PRINT C$
```

Output:

```
    BY T.S. ELIOT
```

Any legal alpha expression, including HEX literal strings, can be concatenated with alpha literals or alpha variables. For example:

```
    100 A$ = "APRIL IS THE CRUELEST MONTH"
    200 C$ = A$ & HEX(2C) & " BREEDING"  /* HEX(2C)="," */
    300 PRINT C$
```

Output:

```
    APRIL IS THE CRUELEST MONTH, BREEDING
```

5.2.3 Relational Operators

Relational operators are used in IF...THEN statements and in the POS function (see Section 5.6, Numeric Functions With Alpha Arguments) to compare values of alphanumeric data.

In IF...THEN statements, the values of two strings are compared one character at a time on the basis of their position in the ASCII collating sequence (see Appendix G). In such a comparison, the first characters of each string are compared. If they are different, the string containing the character of a higher position in the collating sequence is the greater of the two. If they are the same, the second characters of the two strings are compared in the same way; this process is repeated until a pair of unequal characters is found or until one or both strings is exhausted. If the strings are of unequal length, the shorter one is treated as though it had enough trailing blanks to make it the same length as the longer one, and the comparison continues one character at a time. This usually places the shorter string earlier in the collating sequence, since few characters have a lower ASCII value than the space (HEX(20)). If the strings are of equal length and the comparison shows all characters to be the same, the strings are equal.

The relational operators used with alphanumeric data are the same as those used for numeric relations. They have the following meanings when used with alpha data.

Symbol	Sample Relation	Explanation
=	A$ = B$	A$ is at the same position as B$ in the collating sequence.
<	A$ < B$	A$ preceeds B$ in the collating sequence.
<=	A$ <= B$	A$ preceeds or is at the same position as B$ in the collating sequence.
>	A$ > B$	A$ follows B$ in the collating sequence.
>=	A$ >= B$	A$ follows or is at the same position as B$ in the collating sequence.
<>	A$ <> B$	A$ is at a different position from B$ in the collating sequence.

5.3 ALPHA ARRAY STRINGS

An entire alpha array can be treated as a single alpha variable wherever an alpha variable would be allowed. In this case the alpha array is referred to by its name followed by "()" (the same form used for array-designators). The array is treated as a single continuous character string, called an alpha array string, which in memory is equivalent to a row-by-row path through the elements of the array. For example:

```
100 DIM A$(2,2)3
200 A$(1,1)="1":A$(1,2)="2":A$(2,1)="3":A$(2,2)="4"
300 PRINT A$()
```

```
    Output:
       1  2  3  4
```

Although alpha array strings and alpha array-designators look alike, their usage is generally determined by the syntax. There are cases, however, in which both scalars and arrays are allowed. In these cases, an argument such as A$() will always be regarded as an array-designator, never as an array string. The statements in which this can occur are:

```
ACCEPT     CALL     GET
DISPLAY    SUB      PUT
Disk I/O Statements
```

In these cases STR may be used (e.g., STR (A$())) to indicate that the variable is to be treated as an array string and not as an array designator.

5.4 ALPHA EXPRESSIONS AND ALPHA RECEIVERS

5.4.1 Alpha Expressions

An alpha expression is either one or a series of literals, alpha variables, alpha array strings, or alpha functions connected by concatenation operators (&). Alpha expressions can be evaluated in a variety of VS BASIC statements. In the following example, valid alpha expressions are boxed:

$$\texttt{A\$ = }\boxed{\texttt{B\$}}$$

$$\texttt{IDENT\$ = }\boxed{\texttt{NAME\$ \& "/" \& ADDRESS\$ \& "/" \& SOCSEC\$}}$$

$$\texttt{PRINT }\boxed{\texttt{TEMP\$(I)}}\texttt{ ; }\boxed{\texttt{HEX(0B)}}$$

$$\texttt{IF }\boxed{\texttt{STR(PHRASE\$(I))}}\ \underline{>}\texttt{=}\ \boxed{\texttt{KEY(\#1)}}\texttt{ THEN REARRANGE}$$

$$\texttt{FOR K=1 TO LEN(}\boxed{\texttt{CUCUMBER\$()}}\texttt{)}$$

5.4.2 Alpha Receivers

An alpha-receiver is an alphanumeric item into which data can be stored, such as a variable or an array. Alpha-receivers are used wherever a value is "received," e.g., on the left side of a LET statement, in the argument list of a READ statement, etc.

The following are the only legal alpha-receivers in BASIC:

alpha variable (e.g., A$, A$(1,2))
alpha array string (e.g., B$())
STR function* (e.g., STR(A$,1,1))
KEY function (see Section 8.5, Intrinsic File I/O Functions)

* Only when the first argument is an alpha-receiver.

5.5 ALPHANUMERIC FUNCTIONS

Eight BASIC functions return alphanumeric values.

Function	Meaning	Number and type of arguments
ALL(a$)	A character string consisting entirely of characters equal to the first character of a$.	1 alphanumeric.
BIN(x,d)	An alphanumeric string of d characters whose decimal ASCII value is the integer part of x.	1 numeric (x), 1 integer (d) = 1, 2, 3, or 4.
DATE	A six-character string giving the current date in the form YYMMDD.	None.
FS(#n)	A two-character code indicating the file status for the most recent I/O operation involving file #n. (See Section 8.5, Intrinsic File I/O Functions.)	1 file expression.
KEY(#n)	The value of the "key" field for the last record read from file #n. (See Section 8.5, Intrinsic File I/O Functions.)	1 file expression.
MASK(#n)	The value of the two-character alternate key mask for the last record read from file #n. (See Section 8.5, Intrinsic File I/O Functions.)	1 file expression.
STR(a$,n,m)	The sub-string of a$ which begins at the n-th character of a$ and is m characters long.	1 alphanumeric (a$), 2 numeric (n,m).
TIME	An eight-character string giving time of day to the hundredth of a second in the form HHMMSShh.	None.

Further discussion of these functions can be found under their respective entries in Part 2.

NOTE:

The STR function can be used to refer to the entire defined length of an alpha variable, including trailing blanks, by omitting the second two arguments. See Part 2 for examples of this use.

5.6 NUMERIC FUNCTIONS WITH ALPHA ARGUMENTS

VS BASIC supports four functions which take alpha expressions as arguments and return integer values. These are summarized in the following table, and described in more detail below.

Function Name	Sample Expression	Meaning
LEN	LEN(X$)	The current length of the argument.
NUM	NUM(X$)	The number of consecutive characters in the argument, starting at the first, which form an ASCII representation of a valid BASIC number.
POS	POS(X$="$")	The position within the first argument of the first character that satisfies the indicated relation (equal to, less than, greater than) with the second argument.
VAL	VAL(X$,N)	The decimal equivalent of the binary numeric value of the first N characters of X$.

5.6.1 LEN

The LEN function requires an alpha-expression as its argument, and returns an integer value which is the actual length of the argument. The length of a string of all blanks is 1. For example:

```
LEN("ABCDE") -- Returns 5.
LEN(E$) -- Returns the actual length of E$.
LEN(STR(E$)) -- Returns the defined length of E$.
```

Note that the relation LEN(A$&B$)=LEN(A$)+LEN(B$) is always true.

5.6.2 NUM

The NUM function requires an alpha expression as an argument, and returns an integer value equal to the number of sequential characters in the argument that form a legal BASIC floating-point constant. Allowable characters are 0 through 9, E, ., +, -, and space (in the leading or trailing position), provided that they conform to the syntax for floating-point constants.

The count begins with the first character of the argument, and ends with the first character that violates the floating-point syntax. NUM searches the entire defined length of the argument; if no characters are found which violate the floating-point syntax, NUM returns the defined length. If the argument is entirely blank, NUM returns 0. Leading and trailing spaces are included in the count. Thus:

```
NUM ("1E 88") returns 1
NUM ("1E8 8") returns 4
NUM (" 1E8 8") returns 5
```

NUM can be used to validate an alphanumeric representation of a number before attempting to convert it to internal numeric binary form with the CONVERT statement, described in Part II.

NUM will not stop its search after finding more than fifteen digits in the numeric constant, even though subsequent attempts to evaluate that number will ignore all digits other than those belonging to an exponent after the fifteenth significant digit.

Note that NUM does not check the value of a number, only whether it is formatted correctly. Thus NUM("1E88") returns 4, even though 1E88 is greater than the largest allowed floating-point constant.

5.6.3 POS

The POS function requires three components in its argument (not to be separated by commas): (1) an alpha expression, optionally preceded by a minus-sign; (2) a relational operator; and (3) a second alpha expression. The relational operator is taken from the set:

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

The POS function searches the first string for a character which satisfies the specified relation to the first character of the second string. Thus, POS (E$<="*") searches E$ for a character less than or equal to "*".

Comparisons are based on the ASCII-coded values of the characters. Thus, searching a string for a character less than or equal to " " means searching a string for a character whose hexadecimal value is less than or equal to HEX (20), the ASCII value of the space character.

The POS function returns an integer value which is the position in the first expression where the comparison first succeeds. The leftmost position in the expression is named 1; the position to the right of that is 2, and so on. If no character is found within the first expression which satisfies the relation, POS returns a value of 0.

The optional minus sign to the left of the first alpha expression indicates the direction of the search. Normally, searches are left-to-right; if the minus sign is present, the search will proceed from right-to-left. The entire defined length of the expression is searched until either a match is found or the expression is exhausted. POS(E$=" ") returns the position of the leftmost space in E$. POS(-E$=" ") returns the position of the rightmost space.

NOTE:

When comparing alpha string variables with literal strings or other alpha variables (e.g., IF A$ < "ABC"), values are compared character by character. Trailing spaces are considered equivalent to HEX(20) in determining where to place each value in the collating sequence. The values fall at the same location in the collating sequence (i.e., they are equal) even if they do not have the same number of trailing spaces, as long as all their other characters are equal. For example:

```
 100 DIM A$4, B$5, C$5
 200 A$="ABC"
 300 B$=HEX(41424321)  /* HEX(41424321)="ABC!" */
 400 C$="ABC  "
 500 IF A$=B$ THEN 800
 600 IF A$=C$ THEN 1000
 700 PRINT "A$ NOT EQUAL TO B$ OR C$." : GOTO 1100
 800 PRINT "A$=B$: ";A$;"=";B$
 900   GOTO 1100
1000 PRINT "A$=C$: ";A$;"=";C$
1100 END
```

Output:

```
A$=C$: ABC=ABC
```

5.6.4 VAL

The VAL function requires an alpha expression as an argument. A digit whose value is 1, 2, 3, or 4 can be supplied as a second argument; if it is omitted, a value of 1 is assumed for the second argument.

The VAL function extracts up to four characters from the alpha expression, depending on the value of the second argument, and returns a decimal integer which is equivalent to the binary value of the extracted character(s).

VAL(A$) or VAL(A$,1) will simply return the decimal value of the ASCII code for the first character of A$. For instance, VAL("A",1) is 65, VAL("B",1) is 66, and so on. The value will range from 0 through 255. The value returned is the decimal equivalent of character's binary ASCII code, not the hexadecimal value.

VAL(A$,2) will return an integer whose value is:

(code for 1st char.)*256 + (code for 2nd char.)

It will be in the range 0 through 65535.

VAL(A$,3) returns a value between 0 and 16777215:

(code for 1st char.)*65536 + (code for 2nd char.)*256 +
(code for 3rd char.)

VAL(A$,4) computes the following value:

(code for 1st char.)*16777216 + (code for 2nd char.)*65536 + (code for 3rd char.)*256 + (code for 4th char.)

This computation requires all 32 bits of the integer; furthermore, overflow may occur, causing the result to be a negative integer. The value of the result will range between -2147483648 and 2147483647, inclusive.

The BIN function can be used to extract characters from an integer expression containing their binary values, reversing the operation performed by VAL.

5.7 LOGICAL EXPRESSIONS

The alpha operators and functions discussed so far have all involved manipulation of single characters and strings of characters. It is also possible to manipulate individual bits within the bytes which represent stored characters. This is done in a special type of alpha expression called a logical expression, which can be used only on the right-hand side of an assignment (LET) statement.

Logical expressions are alpha expressions containing any of several logical operators and have the general form:

[operator] operand [operator operand] ...

```
where operator is one of   ADD[C]
                           AND
                           OR
                           XOR
                           BOOLh
```

and where operand is an alpha expression or ALL(alpha expression).

Note that concatenation (&) and parentheses are not allowed within logical expressions.

5.7.1 Evaluation of Logical Expressions

A statement of the form "LET alpha-receiver = logical expression" is evaluated as follows:

1. If the expression begins with an operand, the receiver is assigned that operand (i.e., like a simple LET statement).

2. From left to right, the next operator operates on the operand to its right and the receiver (i.e., the receiver is used as an operand). In all cases, the defined lengths of both arguments are used, with the operation proceeding one byte at a time as follows:

   a. AND, OR, XOR, BOOLh -- The operation proceeds one byte at a time from left to right. If the operand is shorter than the receiver, the remaining characters of the receiver are unchanged. If the operand is longer than the receiver, the operation stops when the receiver is exhausted. The specific effects of these operators are described in the next section.

   b. ADD, ADDC -- The operation proceeds one byte at a time from right to left. If the operand and receiver are not the same length, the shorter one is left-padded with hex zeros. The result is right-justified in the receiver, with high-order characters truncated if the result is longer than the receiver. The specific effects of these operators are described in the next section.

3. The receiver always gets the result of the operation; Step 2 is then repeated until all operator-operand pairs are used up.

Part of an alpha variable can be operated on by using the STR function to specify a portion of the variable. For example,

```
100 STR(A$, 3, 2) = ADD B$
```

operates only on the 3rd and 4th bytes of A$.

5.7.2 Logical Operators

In the following examples, assume A$ has a defined length of 2 bytes.

AND -- Logically ANDs the two operands, one byte at a time, as indicated in Table 5-1. For example,

```
            LET A$ = HEX(0F0F) AND HEX(0FF0)
   Result:  A$ = HEX(0F00)
```

OR -- Logically ORs the two operands, one byte at a time, as indicated in Table 5-1. For example,

```
            LET A$ = HEX(0F0F) OR HEX(0FF0)
   Result:  A$ = HEX(0FFF)
```

XOR -- Logically exclusive-ORs the two operands, one byte at a time, as indicated in Table 5-1. For example,

```
            LET A$ = HEX (0F0F) XOR HEX (0FF0)
   Result:  A$ = HEX(00FF)
```

BOOLh-- Performs one of 16 logical operations specified by the value of the hexadecimal digit h. See the entry in Part 2 under BOOLh for a description and examples of these operations.

ADD -- Adds the binary values of the operands, one byte at a time, with no carry propagation between bytes. For example,

```
            LET A$ = HEX(0123) ADD HEX(00FF)
   Result:  A$ = HEX(0122)
```

ADDC -- Adds the binary values of the operands, one byte at a time, with carry propagation between bytes (like two long binary numbers). For example,

```
            LET A$ = HEX(0123) ADDC HEX(00FF)
   Result:  A$ = HEX(0222)
```

Table 5-1. Logical Operations

Logical operator	Operand 1 Bit =	Operand 2 Bit =	Result Bit =
AND (Result = 1 if both operand bits = 1. Otherwise, result = 0.)	0 0 1 1	0 1 0 1	0 0 0 1
OR (Result = 1 if either or both operand bits = 1. Otherwise, result = 0.)	0 0 1 1	0 1 0 1	0 1 1 1
XOR (Result = 1 if one or the other but <u>not both</u> operand bits = 1. Otherwise, result = 0.)	0 0 1 1	0 1 0 1	0 1 1 0

5.8 SUMMARY OF ALPHANUMERIC DATA FORMATS AND TERMS

5.8.1 Alphanumeric Length

1. ACTUAL OR CURRENT LENGTH (in bytes)
   (as determined by LEN function)

   a. Alpha Variable -- Does not include trailing blanks. If all blank, length=1.

   b. Alpha Array String -- Like a single long alpha variable.

   c. Alpha-expression -- Length = sum of actual lengths of the concatenated arguments.

   d. STR function -- Length is the number of characters extracted, including trailing blanks.

   e. KEY function -- Length is the key length specified in SELECT.

f. Literal -- Length is the number of characters within quotes or the number of hexadecimal digit pairs in HEX.

g. FS function -- Length=2.

h. DATE function -- Length=6.

i. TIME function -- Length=8.

j. MASK function -- Length=2.

k. BIN function -- Length as specified by second argument of BIN (1,2,3, or 4; default=1).

2. DEFINED LENGTH

a. Alpha Variable -- As specified in DIM, COM, or most recent MAT REDIM. (Default = 16.)

b. Alpha Array String -- Product of 3 dimensions (e.g. row, column, element length) in DIM, COM, or most recent MAT REDIM. (Default 10 x 10 x 16.)

c. Alpha-expression -- Except alpha variables and alpha array strings. Same as actual length.

d. All Other Alpha Forms -- Same as actual length.

5.8.2 Alphanumeric Terms

```
1.    Alpha Scalar Variable:  letter     letter
                                         digit          ...  $
                                         underscore
        (not to exceed 64 letters, digits, and underscores)

2.    Alpha Array Name:  letter     letter
                                    digit               ...  $
                                    underscore
        (not to exceed 64 letters, digits, and underscores)
```

3. <u>Alpha Array-designator</u>: letter $\left[\left\{\begin{array}{l}\texttt{letter}\\ \texttt{digit}\\ \texttt{underscore}\end{array}\right\}\right]$ ... \$()

(not to exceed 64 letters, digits, and underscores)

4. <u>Alpha Array Element</u>: letter $\left[\left\{\begin{array}{l}\texttt{letter}\\ \texttt{digit}\\ \texttt{underscore}\end{array}\right\}\right]$ ... \$(exp[,exp])

(not to exceed 64 letters, digits, and underscores)

5. <u>Alpha Variable</u>: $\left\{\begin{array}{l}\texttt{alpha scalar variable}\\ \texttt{alpha array variable}\end{array}\right\}$

6. <u>Alpha Array String</u>: letter $\left[\left\{\begin{array}{l}\texttt{letter}\\ \texttt{digit}\\ \texttt{underscore}\end{array}\right\}\right]$ ... \$()

(not to exceed 64 letters, digits, and underscores)

(Treated as a <u>single long alpha variable</u>)

7. <u>Literal</u>:

$$\left\{\begin{array}{l}\texttt{"}\left\{\begin{array}{l}\texttt{any}\\ \texttt{character}\\ \texttt{except}\\ \quad\texttt{"}\end{array}\right\}\left[\left\{\begin{array}{l}\texttt{any}\\ \texttt{character}\\ \texttt{except}\\ \quad\texttt{"}\end{array}\right\}\right]\texttt{"}\ldots\\ \\ \left\{\begin{array}{l}\texttt{any}\\ \texttt{character}\\ \texttt{except}\\ \quad\texttt{'}\end{array}\right\}\left[\left\{\begin{array}{l}\texttt{any}\\ \texttt{character}\\ \texttt{except}\\ \quad\texttt{'}\end{array}\right\}\right]\ldots\end{array}\right\}$$

HEX(hh[hh]...)

8. <u>h</u>: a hex digit (0,1,2,...,9,A,B,C,D,E, or F)

9. <u>Alpha Receiver</u>: $\left\{\begin{array}{l}\texttt{alpha-variable}\\ \texttt{STR(alpha receiver[,[exp][,exp]])}\\ \texttt{alpha array string}\\ \texttt{KEY (file-expression [,exp])}\end{array}\right\}$

10. <u>Alpha Expression</u>:
(or <u>Alpha-exp</u>)

$$\left\{\begin{array}{l}\texttt{alpha-receiver}\\ \texttt{literal}\\ \texttt{DATE function}\\ \texttt{TIME function}\\ \texttt{BIN function}\\ \texttt{MASK function}\\ \texttt{FS (file-expression)}\\ \texttt{alpha-exp \& alpha-exp}\\ \texttt{(alpha-expression)}\\ \texttt{STR (alpha-exp[,[exp][,[exp]]])}\end{array}\right\}$$

11. Logical Expression:

```
[operator] operand [operator operand] ...
where:
                    ⎧ADD[C]⎫
                    ⎪AND   ⎪
  operator =        ⎨BOOLh ⎬     operand =   ⎧alpha-expression⎫
                    ⎪OR    ⎪                 ⎩ALL function    ⎭
                    ⎩XOR   ⎭
```

5.8.3 Alphanumeric Operations

1. The following applies to alpha values used in any BASIC functions or operations:

   a. In statements that can alter the values of variables (e.g., LET, COPY), values of alpha expressions which are not acting as receivers are copied to a temporary location*; the value in the temporary location is then used in whatever operations are specified. This includes alpha receivers enclosed in parentheses.

      Alpha receivers, on the other hand, are never moved, but are operated on in place, except in the TRAN statement (described in where Part 2). The differences in results which may occur depending upon whether or not an expression is a receiver are most apparent in the following:

      ```
      multiple assignment (LET) statements
      LET statements incorporating
          ADD
          AND
          OR
          XOR
          BOOLh
      COPY statements
      ```

      For example,

      ```
      100 LET A$="A"
      200 LET B$="B"
      300 LET A$, B$, C$ = A$ & B$  /* THIS IS A MULTIPLE "LET" */
      400 PRINT A$,B$,C$
      ```

      prints:

      ```
      AB                AB                AB
      ```

* An expression is said to be "acting as a receiver" in the context of a particular statement if it is syntactically a receiver (see Section 3.2, Constants, Variables, Receivers, and Expressions) and is also being assigned a new value in that statement.

When statement 300 is executed, the value of A$ ("A") is concatenated with the value of B$ ("B") and the result ("AB") stored in a temporary location. This string is then copied from the temporary location into A$, B$, and C$ sequentially. If a temporary location were not used, statement 300 would be equivalent to

```
300 LET A$ = A$ & B$ : LET B$ = A$ & B$ : LET C$ = A$ & B$
```

and the program would print

```
AB            ABB            ABABB
```

b. In general, any operation requiring character comparison or movement is done one character at a time. This applies to each of the functions listed above.

2. TRAN always moves the translation alpha expression to a separate translate table inaccessible to the user. Thus, TRAN may never translate its own table.

3. Statements which perform multiple assignments always assign values from left to right. This applies particularly to LET, INPUT, ACCEPT, GOSUB'(), READ, and GET. This can be an important consideration, especially when receivers in the same location are specified more than once in the receiver list.

CHAPTER 6
CONTROL STATEMENTS

6.1 INTRODUCTION

A VS BASIC program is normally executed in ascending line-number sequence, with multiple statements on a line executed from left to right. VS BASIC also provides a number of statements, called control statements, which can be used to alter the normal sequence of execution.

CALL	FOR...NEXT	INPUT (some cases)
RETURN	IF...THEN...ELSE	ACCEPT (some cases)
END	ON...GO TO	STOP
GOSUB	ON...GOSUB	Unusual condition and error/data conversion exit clauses for some statements
GOSUB'	GOTO	

Figure 6-1. BASIC Control Statements

Control statements provide BASIC with the following facilities:

1. Halting Execution -- END, if encountered in a program, terminates program execution and returns control to the Command Processor or the invoking program or procedure. If encountered in a subroutine, END returns program control to the calling program (see Number 3). STOP temporarily halts execution until the user presses the workstation ENTER key or, under defined conditions, one of the program function keys.

   INPUT, ACCEPT, and STOP temporarily halt execution to enable the program user to supply the program with run-time data, or, under defined conditions, to press a program function key. These statements are discussed in Chapter 7 and under their separate entries in Part 2.

2. Unconditional Program Branching -- GOTO transfers control to the line number or statement label specified by the GOTO statement. The GOTO statement is discussed under its entry in Part 2.

3. Conditional Branching -- IF...THEN...ELSE enables the program to test a relationship -- the operand of the IF clause -- and branch according to the result of the test. If the relationship is true, the THEN clause is executed and the ELSE clause is not. If the relationship is not true, the ELSE clause (or in the absence of an ELSE clause, the next sequential executable statement) is executed and the THEN clause is not. The IF statement is discussed under its entry in Part 2.

4. Branching to Subroutines -- GOSUB, GOSUB', and CALL transfer control to various kinds of subroutines, after their execution, control can be returned to the main body of the program by RETURN or END. GOSUB and GOSUB' are discussed under their entries in Part 2; Subroutines are discussed in Section 6.3, Subroutines; Section 6.4, Internal Subroutines; Section 6.5, External Subroutines; and under the entries for the various statements in Part 2.

5. Looping -- A useful feature of BASIC is its ability to repeatedly execute a defined section of code. This section of code is called a loop. BASIC provides a pair of statements, FOR and NEXT, that automatically mark a loop and determine the number of times it will be executed. FOR and NEXT are discussed under their entries in Part 2.

6. Unusual Condition Exits -- VS BASIC provides a number of exits for data error and end-of-data conditions which would otherwise result in termination of a program. These include the DATA, IOERR, and EOD (end-of-data) clauses in the file I/O and CONVERT statements. These clauses in file I/O statements are discussed in Section 8.6, Error Recovery, and in the appropriate entries in Part 2.

6.2 STATEMENT LABELS

Any statement in a VS BASIC program may be identified by a statement label, which immediately precedes it. A statement label (or simply a "label") may be any string of up to 64 letters, digits, and underscores, provided that the first character is a letter and that the string is not a VS BASIC reserved word (see Section 2.2, Statements, and Appendix A). Using labels, a programmer can write statements which alter the flow of program execution without having to keep track of line numbers. For example, instead of writing GOTO 100 (where 100 is a program line number), the programmer can write GOTO PART2, where PART2 is a statement label. In this case execution continues with the first executable statement following the label PART2. A label can occur alone on a line, or at the beginning, middle, or end of a line containing one or more statements. If a label is followed by one or more statements on the same line, the label and the following statement must be separated by a colon. If a label occurs alone on a line or at the end of a line, the colon is optional.

The following are some examples of correct and incorrect usage of statement labels:

CORRECT:

1.
```
500 PART2
600 PRINT "ENTER DATA FOR PART 2"
```
(Label is PART2.)

2.
```
900 FIRST_TIME : RAISIN=RAISIN+1 : RETURN
```
(Label is FIRST_TIME.)

3.
```
100 LET CAT=(10*X)/#PI : FIRST : READ Z
```
(Label is FIRST.)

4.
```
300 READ NAME$, STREET$, PHONE$ : HENRY
350 EXCH$=STR(PHONE$, 5, 3)
```
(Label is HENRY.)

INCORRECT:

1.
```
200 NEXT : IF KRISP = 99 THEN 6100
```
(NEXT is the verb of an executable statement used to terminate a FOR...NEXT loop and is thus a reserved word. Reserved words cannot be used as labels.)

2.
```
700 LAST ONE
800 FOR I=1 TO 100 : READ FRED(I) : NEXT I
```
(LAST ONE: labels cannot contain embedded spaces. The compiler would interpret this as being a single label (LAST) and would expect it to be followed by a statement terminator (a colon or the end of the source program line).)

3.
```
400 LET B(J)=SQR(X(J)) : LABEL PRINT B(J)
```
(LABEL should be separated from the following statement (PRINT) by a colon.)

4)
```
1000 CAT&MOUSE : IF CS>MS THEN 1700
```
(CAT&MOUSEcontains a character ("&") which is not a letter, number, or underscore and is thus invalid as a label.)

5)
```
5200 2ND_TIME : GOSUB 7520
```
(2ND_TIME: the first character of any label must be a letter.)

6.3 SUBROUTINES

A subroutine is a group of program lines which can be invoked from any point in a program to perform a specific task. When execution of a subroutine is completed, processing normally returns to the point in the program from which the subroutine was invoked. The same set of instructions can be accessed from many different points in a program, with control returning (if desired) to the part of the program that called the subroutine.

VS BASIC provides internal and external subroutines. Internal subroutines are included as part of the code in the main BASIC source file. They are invoked by a GOSUB or GOSUB' statement or, under certain circumstances, by pressing an appropriate PF key while execution is halted by INPUT or STOP. Subroutines invoked by GOSUB' or a PF key are marked in the source file by a DEF FN' statement. Subroutines invoked by GOSUB need not be marked -- GOSUB transfers control to a specific line number or statement label.

External subroutines are written as independent files, beginning with a SUB statement. After compilation, they are linked to the main program via the LINKER utility (see Subsection 1.4.3, The LINKER Utility, Subsection 6.5.3, Compiling, Linking, and Running, and the VS Programmer's Introduction). The main program invokes external subroutines by means of the CALL statement. External subroutines can be linked to any number of calling programs, making them a useful way to code routines that may be used by more than one program. An external subroutine has to be coded only once. If it is later changed, it has to be recompiled only once, and the calling programs do not have to be modified.

6.4 INTERNAL SUBROUTINES

VS BASIC provides three ways of invoking an internal subroutine: GOSUB, GOSUB', and the execution-time pressing of program function keys. A brief summary follows; a full discussion of each statement can be found under the appropriate entry in Part 2 of this manual.

6.4.1 GOSUB Subroutines

The GOSUB statement branches to a line number or a statement label. For example:

```
500 GOSUB 2000                  900 GOSUB RABBIT
```

When executed, statement 500 transfers control to line 2000; statement 900 transfers control to the statement labeled RABBIT. The beginning of the subroutine need not be specially marked. Any valid BASIC statement can begin a GOSUB subroutine. For example:

```
2000 REM THIS SUBROUTINE PRINTS THE CURRENT VALUE OF A
2100 PRINT "A="; A
2200 RETURN
```

When a GOSUB is executed, the program stores the location of the statement which invoked the subroutine. At the end of the subroutine, marked in this case by a RETURN statement, execution continues at the statement following the GOSUB statement on line 500. If the same subroutine is subsequently invoked from line 1200, execution continues then at the statement following the GOSUB statement on line 1200. The end of a GOSUB subroutine is marked by a RETURN or RETURN CLEAR. RETURN CLEAR causes execution to continue with the statement following RETURN CLEAR, instead of returning to the statement after the GOSUB.

6.4.2 GOSUB' Subroutines

The GOSUB' statement branches to a subroutine which is marked by a DEF FN' statement. For example:

```
500 GOSUB'112
```

This statement causes control to pass to the statement DEF FN'112. The range of allowable DEF FN' numbers is 0 to 255. Following execution of the marked subroutine, control is returned to the statement following the GOSUB' by a RETURN, or to the statement following the subroutine by a RETURN CLEAR.

The most important difference between GOSUB and GOSUB' subroutines is that the latter allow the passing of an argument list from the main program to the subroutine. This is useful where a subroutine may be called from different parts of a program to perform the same series of operations on different variables. For example, suppose one wanted to write a subroutine which would add some variable to the length of an alpha variable. The subroutine could be written as:

```
5000 DEF FN'100 (STRING$, COUNT%)
5100 PIGEON% = LEN(STRING$) + COUNT%
5200 RETURN
```

The subroutine might be called from elsewhere in the program to perform its operation on a string called PHONE$ and an integer called BOOK% by the statement

```
400 GOSUB'100 (PHONE$, BOOK%)
```

When line 400 is executed, implicit assignment statements are performed which assign the current value of PHONE$ to STRING$ and that of BOOK% to COUNT%. By this means, the arguments are "passed" to the subroutine. When the subroutine is completed (by the execution of the RETURN on line 5200), the sum of the value of BOOK% and the length of PHONE$ is stored in PIGEON%.

The same subroutine might be called again from a different point in the program to operate on two different variables:

```
1500 GOSUB'100 (BOX$, CAR%)
```

Once again, the result would be stored in PIGEON%.

While this may appear similar to the scheme of dummy variables used in defining user-defined functions (see Subsection 4.4.2, User-Defined Functions), there is one very important difference. Dummy variables in function definitions have no significance beyond the function definition itself. Thus the function may use as a dummy variable a name which is also used as a regular ("non-dummy") variable elsewhere in the program, without affecting the value of that variable. The arguments used in defining a DEF FN' subroutine, however, are regular variables which are not in any way distinct from those used in the main program. If a variable used in a main program is also used in a DEF FN' subroutine, the original value of that variable is lost when the DEF FN' subroutine is called. Consider the following examples:

Program 1:

```
    100 DUM=1
    200 X=16
    300 DEF HALF_ROOT(DUM)=SQR(DUM)/2    /* FUNCTION DEFINITION */
    400 Y=HALF_ROOT(X)                   /* SUBROUTINE CALL */
    500 PRINT DUM, X, Y
```

Output:

```
    1               16                2
```

Program 2:

```
    100 DUM=1
    200 X=16
    300 GOSUB'100 (X)        /* SUBROUTINE CALL */
    400 PRINT DUM, X, Y
    500 DEF FN'100 (DUM)     /* BEGINNING OF HALF-ROOT SUBROUTINE */
    600 Y=SQR(DUM)/2
    700 RETURN               /* END OF SUBROUTINE */
```

Output:

```
    16              16               2
```

In the first program, DUM is used in defining the function HALF_ROOT, which is then called to operate on the value of X. Afterwards, DUM retains the value it was assigned in line 100. In the second program, DUM is again used to define the same operation (lines 500, 600). When the subroutine is called in line 300, however, DUM is assigned the value of X, thus destroying the value originally assigned in line 100.

Arguments are passed in the exact order in which they appear in the argument lists -- the first item in the GOSUB' list to the first item in the DEF FN' list, the second to the second, and so on. Arguments must correspond in type; an alphanumeric argument cannot be passed to a numeric receiver, and vice versa. Floating-point arguments may, however, be passed to integer receivers, and vice versa.

6.4.3 Program Function Keys

The VS workstation has, at the top of the keyboard, 16 Program Function (PF) keys, each of which can be pressed with or without the SHIFT key for a total of 32 program functions. BASIC can program any of the PF keys to invoke the marked subroutines.

Subroutines invoked from the keyboard are marked by DEF FN' statements, with the restriction that the DEF FN' numbers for subroutines accessible by the PF keys must be between 1 and 32 (instead of 0 to 255, as with the GOSUB' statement). A DEF FN' subroutine can be invoked from the keyboard whenever execution has been temporarily halted by a STOP or INPUT statement. At this time, depressing a PF key will cause control to pass to the DEF FN' subroutine corresponding to that PF key. For example:

```
     500 STOP
          .
          .
          .
    2000 DEF FN'1
```

Pressing PF 1 when execution is halted by the STOP at line 500 invokes the subroutine marked by DEF FN'1. Keying ENTER causes the normal sequence of execution to continue with the statement following STOP. Depressing a PF key for which there is no corresponding DEF FN' subroutine in the program causes the workstation alarm to sound and the key is ignored.

Keyboard subroutines operate in the same manner as GOSUB' subroutines, with one exception. A RETURN statement passes control back to the STOP or INPUT statement, instead of to the following statement. Thus, DEF FN' subroutines can be invoked repeatedly from a STOP or INPUT statement.

NOTE:

To avoid unintended transfers to marked subroutines, it is recommended that numbers 1 - 32 be used only for those subroutines meant to be invoked by PF keys.

6.5 EXTERNAL SUBROUTINES

A second class of subroutines are not contained in the body of the program (the same file), but instead reside in a separate file. Such subroutines, referred to as "external subroutines" or "subprograms" are defined with the SUB statement and invoked with the CALL statement. In general, a BASIC source file can contain either a main program or subprogram. Subprograms are distinguished by the fact that their first statement, other than REM, must be the SUB statement.

6.5.1 Operation of External Subroutines

The SUB statement declares a program to be a subroutine and specifies the subroutine name, allowing it to be referenced in CALL statements. The CALL statement transfers control from one program (called the calling program) to the beginning of another program (the external subroutine), which is referenced using the name specified in the SUB statement. The point at which the CALL statement occurs in the main program is saved, so that control may later return to that point. A subroutine may contain one or more CALL statements, by which it calls other subroutines. A subroutine cannot call itself.

When control is passed to an external subroutine by a CALL statement, the normal sequence of execution is followed in the subroutine until an END statement is encountered in the subroutine. Control then returns to the statement following the last CALL statement executed.

In a calling program, a CALL statement invokes an entire sequence of statements, in whatever order they are contained in the subroutine, without affecting the overall flow of control in the calling program.

In both form and operation, external subroutines are self-contained programs; they must, however, begin with the SUB statement and, in some cases, operate on values obtained from the calling program. Once the external subroutine has been called, the only way execution can pass back to the calling program is by the execution of an END statement. All branching instructions (GOTO, IF...THEN...ELSE, GOSUB, GOSUB', etc.) in an external subroutine refer to line numbers or statement labels within that subroutine. For example, it is not possible to GOTO a statement outside the subroutine. Note that this differs from internal subroutines, which may branch to any portion of the calling program.

6.5.2 Form of External Subroutine Calls and Definitions

An external subroutine is any BASIC program having a SUB statement as its first statement (other than REM). The general form of the SUB statement is:

```
SUB "name" [ [ADDR] (arg[, arg] ... ) ]
```

where "name" is the name of the subroutine, consisting of any string of 1 to 8 alphabetic or numeric characters (including @, #, and $). The presence or absence of the word ADDR specifies the way in which the optional arguments are to be passed between the calling program and the subroutine.

An external subroutine is called by a CALL statement in another program. The general form of the CALL statement is:

```
CALL "name" [ [ADDR] (arg[, arg] ... ) ]
```

where "name" is the name specified in the SUB statement of the subroutine being called. Again, the word ADDR and the optional argument list specify the form of argument passing to be used.

NOTE:

The name of the subroutine is defined by the literal in the SUB statement, not by the name of the file containing the subroutine. These two names need not be the same.

6.5.3 Compiling, Linking, and Running

Each main (calling) program and each external subroutine is entered into a separate source file using the EDITOR. The BASIC compiler must be called separately for each program and external subroutine to produce an object file for each one. Thus, in writing a program which used two external subroutines, three source files would be created (one for the calling program and one for each of the two subroutines). The BASIC compiler would then be run three times, once to compile each of these files, resulting in three object files.

Before the program and subroutines can be run, the object programs must be linked together. Linking is the process of merging multiple object modules into one. Linking programs and external subroutines is done by a VS utility program called LINKER, which is run from the VS Command Processor. LINKER resolves subroutine name references between object modules to produce a single object module which can then be run. LINKER asks the user for the names of all the object files to be linked together and then requests a name for the single output file. The files are then linked and the final output file generated.

The original program and subroutines can then be run from the Command Processor as one would run any other program, using the program name specified as the output file for LINKER. For further details on how to use LINKER, see the VS Programmer's Introduction.

6.5.4 Passing Values to External Subroutines

Since calling programs and external subroutines are written and compiled as separate programs, there must be a way of passing data between them if subroutines are to process any of the data used in a calling program. In BASIC, values are passed in one of two ways:

1. The values may be made arguments of the subroutine. An argument of a subroutine is a value which may be operated on by action of the subroutine. Arguments are enclosed in parentheses following both the CALL and the SUB statements.

2. The values may be stored as common variables (or "placed in common"). Common variables are variables that are stored in a particular area of memory accessible to all programs and subroutines that are run together. Variables are placed in common using the COM statement (see Subsection 6.5.4, Passing Values to External Subroutines and the COM statement entry in part 2).

Arguments

The arguments in a SUB statement must be either variables, array designators, or file numbers. These arguments are dummy variables (like those in user-defined function definitions; see Subsection 4.4.2, User-Defined Functions) which indicate the names which will be used in the subroutine to refer to the arguments specified by any particular CALL to that subroutine. The actual names used for dummy variables are significant only within the subroutine and need have no connection with names used in a calling program, except for type correspondence, described in Subsection 6.5.6, Argument Types.

Arguments of a CALL statement must be numeric or alpha expressions, or file expressions. Their values are passed one by one to the dummy variables in the SUB statement of the external subroutine at run time. The value of the first expression in the CALL argument list is passed to the first dummy variable in the SUB statement, the second to the second, and so on.

Values are passed back to the calling program when the subroutine ENDs, provided the arguments in the CALL statement are receivers. A subroutine cannot manipulate or examine any value used in the calling program unless it is passed through an argument list or common storage. For example:

Calling program:

```
100 A=10 : X=500
200 CALL "DOUBLE"(A)
300 PRINT A, X
```

Subroutine:

```
100 SUB "DOUBLE"(X)
200 X=2*X
300 END
```

Output:

```
20            500
```

Note that although both programs use variables called X, only the value of A is passed to the subroutine's X since it is the only argument specified in the CALL statement. The subroutine's X is a dummy variable which, in this case, is temporarily assigned the value of A. The subroutine doubles the value of the argument (in this case, A) and then passes this value back to the calling program when the subroutine ends. The value of the variable called X in the calling program remains unchanged.

When an array or an alphanumeric value is used as an argument of an external subroutine call, BASIC normally passes a descriptor of the value to the subroutine, rather than the actual value. A descriptor is a set of data which specifies:

1. The type of the argument (alpha scalar, alpha array, integer array, or floating-point array).

2. The length of the value if it is alphanumeric (element length if an alpha array).

3. The dimensions of the argument if it is an array.

4. The address in memory at which the value is stored (a "pointer" to the value).

This scheme of passing descriptors between calling programs and subroutines is normally used when BASIC subroutines are called from BASIC programs. Subroutines and calling programs written in other languages (e.g., COBOL, Assembler) use a different scheme in which only the address of the value is passed. To enable BASIC programs and subroutines to be linked and run with programs and subroutines written in either BASIC or some other language, two forms of CALL and SUB are available.

1. Non-ADDR Form -- The standard BASIC argument-passing scheme that passes/accepts the descriptors constructed for arrays and alpha-expressions. With this form, any dimensions or lengths specified within the SUB program are ignored, since they are specified by the descriptors. Only the vector/matrix/scalar distinction is significant. Examples:

```
700 CALL "INVOICE" (PN%(), Q%())

100 SUB "INVOICE" (PART_NO%(), QUANTITY%())
```

2. ADDR Form -- Generally used when either the calling program or the subprogram is non-BASIC; Its effect differs depending on the statement in which it is used:

   CALL: The ADDR form of CALL causes all argument-passing to be done via pointers to the actual values; descriptors are not constructed. This method of argument-passing will properly pass arguments to non-BASIC (e.g., COBOL) programs, which always assume that there are pointers directly to the data. Example:

```
900 CALL "PLOT" ADDR (H$, V$)
```

SUB: The ADDR form of SUB causes the program to assume that argument-passing was done as described in CALL i.e., without descriptors. (Such CALLing may have been done from a COBOL program, for example.) However, this implies that the dimensions and lengths used must be those specified within the SUB subroutine. Thus, these dimensions and lengths (or defaults, if omitted) are significant, unlike in the non-ADDR form. Example:

```
100 SUB "PLOT" ADDR (X$, Y$)
200 DIM X$ 100, Y$ 100
```

NOTE:

Languages other than BASIC use different internal formats for representing numeric data. Numeric data to be passed between BASIC and non-BASIC program modules must be converted to the appropriate format. Programmers planning to write BASIC programs or subprograms which call or are called by programs in other languages should see Appendix D for information on compatability of numeric data formats and data conversion routines.

Common Variables

Certain variables may be made accessible to all programs and subroutines that are linked and run together by use of the COM statement. A COM statement must appear in all of the programs and subroutines which are to be run together if the programs are to manipulate or examine any of the same data. The COM statement in each program must precede any reference to any variable which is to be stored in common.

The COM statement consists of the word COM followed by a list of alpha or numeric scalar or array names. Array names may be followed by one or two integers in parentheses giving the dimensions of the array. If these dimensions are omitted, the default dimensions assumed are 10 rows by 10 columns. Alphanumeric scalars or array names may be followed by an integer giving the length, in bytes, of the scalar or the elements of the array. If the length imdicator is omitted, a default length of 16 bytes is assumed. For the general form of the COM statement, see Part 2.

As is true with passed arguments the names used for common variables in a subroutine need not correspond with those used in the calling program, except in type (e.g., integer, alpha, array, etc.). Common variables referenced in calling programs and subroutines are associated with each other by their position in the COM statements. This means that, in many cases, a subroutine may require a COM list which includes variables not actually used by the subroutine, simply to indicate where in the common area certain needed variables are stored.

For example, in

```
100 COM A$5, B%, C(100), D
200 CALL "SUB1"
300 CALL "SUB2"
```

suppose that "SUB1" is a subroutine which performs some operation on the 100-element array called C() in the main program, and that "SUB2" operates on the other variables in the COM list (line 100). Even though "SUB1" does not need to access A$ and B%, they must be accounted for in a COM statement so that the subroutine can find the 100-element array in the common area. If

```
100 SUB "SUB1"
200 COM X(100)
```

was written, the subroutine would look for the 100-element array, called X() in the subroutine, at the beginning of the common area. In fact, the first item in the common area is a 5-byte character string, called A$ in line 100 of the calling program. The subroutine would read the beginning of A$ as the beginning of its 100-element array, which would not produce the intended results when the program is run. A correct form for the subroutine's COM statement is

```
200 COM M$5, N%, X(100)
```

In this case, even though "SUB1" actually needs to use only array X(), it will look for it after a 5-character alpha string and an integer in the common area. The last item in the common area, a floating-point variable called D by the calling program, need not be specified in the subroutine COM statement since it occurs in the common area _after_ the only variable needed by the subroutine.

> _NOTE_:
>
> No variable name occuring in the argument list of a SUB statement may occur as another argument of the same SUB statement, nor in a COM statement in that subroutine. However, calling programs may pass common variables to a subroutine as arguments in a CALL statement.

6.5.5 _Initialization of Subroutine Variables_

The variables in the argument list will receive their arguments from the calling program when the subroutine is called. All other variables (local variables) are initialized when the BASIC program is first executed. String variables are initialized to all spaces; integer and floating-point variables are initialized to zero. However, this initialization occurs only once in the execution of a BASIC program.

Local variables are not reinitialized on subsequent calls. One application of this feature is as follows:

```
 100 SUB "HOOPOE"(arg,arg,...)
 200 REM Let I be a variable which is not in the argument
 250 REM list above.
 300 IF I<>0 THEN 700
 400 REM  Place here statements which are to be
 450 REM  executed only the first time the subroutine
 500 REM  is ever called.
 600 LET I=1
 700 REM  The subroutine continues.
 .
 .
 .
9900 END
```

The first time the subroutine is executed, the variable I is set equal to zero (as are all others not in the argument list). When line 300 is executed, the condition of the IF statement is not satisfied, and execution proceeeds through lines 400 and those following. In line 600, the value of I is set to 1. On all subsequent calls to the subroutine, I will retain this value; when line 300 is executed during a subsequent subroutine call, the IF condition will be satisfied and the statements between line 300 and 700 will be skipped.

6.5.6 Argument Types

As discussed above, the name of an argument passed to a subroutine is not significant in making the connection between calling-program variables and subroutine variables. What is significant is the argument's position in the argument list or common block. Thus, the variable name listed first in the parentheses in the SUB statement or first in a COM list will be the name used by the subroutine to refer to the first argument passed by the CALL statement or specified in the calling program's COM statement. The second variable name will be linked to the second argument in the CALL statement or COM list, and so on. For example:

```
15700 CALL "DANAUS"(A,B,C,D,E)

  100 SUB "DANAUS" (I,J,K,A,B)
```

In this example, the variable name A in the subroutine refers to the variable D in the calling program. If the subroutine intends to access the calling program's variable A, it must use the symbol I. The same is true if variables are passed through common storage:

```
 100 COM A, B, C, D, E
  .
  .
  .
2300 CALL "PLUMBER"

 100 SUB "PLUMBER"
 200 COM I, J, K, A, B
```

If a receiver is placed in the argument list of a CALL statement, the subroutine may transmit a value back to the calling program by assigning a value to the corresponding variable in the argument list of a SUB statement. However, an expression of arbitrary complexity may appear in the argument list of the CALL statement. If the expression is not a receiver, the subroutine may not return a modified value for that argument to the main program. The subroutine may use the corresponding variable from the SUB statement as a receiver; doing so will produce the usual effects during the duration of that call to the subroutine, but no detectable effects after the subroutine returns to the calling program.

For example, constants, literals, and complex expressions may occur in the argument list of a CALL statement. This precludes the possibility of the subroutine returning a value to the calling program by the use of that particular element.

Whether a receiver or an expression occurs as an argument in a CALL statement, its type must match the type of the corresponding argument in the SUB statement it calls.

If the n-th argument of a SUB statement is...	...then the n-th argument of any CALL statement that calls it must be...
an alpha scalar, such as: X$	an alpha-expression.
an integer scalar, such as: X%	an integer expression.
a floating-point scalar: X	a floating-point expression.
an array designator: X$()	an array-designator of the same type (integer, string, floating-point).
a file-number: #3	a file-expression SELECTed by the calling program or passed to it as a parameter.

Note that BASIC will not implicitly convert a numeric quantity in a CALL statement from integer to floating-point, or vice versa, to make its type match the type in the argument list of a SUB statement.

Entire arrays may be passed from a calling program to a subroutine. Only the array-designator-- for example, E() or M$()-- is used as an argument in the CALL statement. The SUB statement must contain, in the corresponding position, an array-designator of the same type (floating-point, integer, or string) as the designator in the CALL statement. The designator used in the SUB statement declares the name by which that array will be referenced in the subroutine. Subroutines can also access arrays used by the main program if the array is declared in COM statements in both programs, as with scalar variables.

An alpha array string (see Section 5.3, Alpha Array Strings) cannot be passed to a subroutine in the usual manner. If the array string M$() occurred as an argument in a CALL statement, it would be interpreted as an array-designator for the array M$, and not as the array string. An array string may be passed to a subroutine by using the expression STR(M$()) as an argument in the CALL statement.

<u>NOTE</u>:

Array strings longer than 256 bytes will be truncated.

The number of subscripts associated with a variable must be consistent between the calling program and the subroutine. If the array passed is two-dimensional (a matrix), it must be used as a matrix in the subroutine. If it is one-dimensional (a vector), it must be used as a vector in the subroutine. A DIM statement should appear in the subroutine to declare each array argument as either a vector or a matrix. In the DIM statement, the supplied dimensions are irrelevant; the actual upper limits are those specified in the array descriptor passed from the calling program. In fact, a MAT REDIM statement (see Subsection 9.2.4, Array Dimensioning) may occur in a subroutine, and the redimensioning of the matrix will remain in effect when control returns to the calling program, unless the subroutine is ADDR type (see Subsection 6.5.4, Passing Values to External Subroutines). In that case, the effects of the MAT REDIM last only until control returns to the calling program.

<u>NOTE</u>:

If an array whose designator does appears in the SUB statement does not appear in a DIM statement in the subroutine, it is assumed to be a matrix.

A file-expression may be passed from a calling program to a subroutine. For instance, if CALL "SUBROU"(#2) calls SUB "SUBROU"(#1), then the subroutine may perform input and output on file #1 (e.g., READ #1 or WRITE #1). The actual file used will be the file which the calling program refers to as #2. Unless linkage is made in this manner, any file selected by the calling program will be inaccessible to the subroutine, and any files selected by the subroutine will be inaccessible to the calling program. Files may be selected by the subroutine whether or not a file with the same number has been selected by the calling program.

6.5.7 Use of External Subroutines

External subroutines might be preferred over internal GOSUB or GOSUB' subroutines. The reasons for this are:

1. A program may be more manageable when broken down into separate subroutines in separate files. Division into subroutines may reflect the logical division of function within a program.

2. A file containing a subroutine may be linked in with several different main programs if the subroutine performs a task common to all the main programs. If changes are made to the subroutine, there is only one copy of the source file for that subroutine which has to be updated. None of the source files for the main programs have to be modified.

3. BASIC programs may call subroutines written not only in BASIC, but also in other languages; subroutines can also be written in BASIC to be called by programs in other languages. Thus, the CALL and SUB statements form BASIC's primary interface to other languages, such as COBOL and Assembler.

CHAPTER 7
WORKSTATION AND PRINTER INPUT/OUTPUT

7.1 INTRODUCTION

VS BASIC contains a group of statements to facilitate I/O operations to the workstation and printer. These statements provide the capability to receive and validate operator-entered data from the workstation, and to create formatted screen output for display at the workstation and formatted print output for the printer. (VS BASIC also supports output to the printer through printer files. See Subsection 8.2.2, File Types, for a discussion of printer files.)

7.1.1 Output

The statements intended purely for data output are:

PRINT -- Used to print data on the printer, or display data at the workstation, one line at a time. The output device is determined by a SELECT statement. The data can be directed to specific positions on the workstation screen with the AT clause, or can be formatted with a USING clause and an auxiliary formatting statement (see Section 7.4, The USING Clause and Format Control Statements). The screen is not cleared before the data are displayed. For a general description of the PRINT statement, see Part 2.

DISPLAY -- Used to output a formatted display to the workstation, using the entire screen. DISPLAY clears the screen before beginning data output so that the new display is constructed only of the contents of the DISPLAY statement. The output of DISPLAY is intended only for the workstation screen, and cannot be directed to the printer. For a description of the operation of the DISPLAY statement, see Section 7.4, The DISPLAY Statement.

All VS BASIC input statements can also be used to some extent to output data or messages to the workstation. None of this output, however, can be directed to the printer. The INPUT and STOP statements can each be used to output a one-line message, with no control over data format or position on the screen. The ACCEPT statement can also output an entire screen of data and literal messages in the same manner as DISPLAY. For descriptions of the INPUT and STOP statements, see Part 2 of this manual. ACCEPT is discussed in Section 7.5, The ACCEPT Statement.

7.1.2 Input

The statements used for data input are:

INPUT -- Used to receive data entered from the keyboard on a line-by-line basis. A message can be inserted in an INPUT statement, to be displayed before the question mark INPUT automatically displays. PF keys can be used in response to an INPUT statement to initiate a branch to a marked subroutine (see Subsection 6.4.3, Program Function Keys).

ACCEPT -- Used to create a formatted display using the entire screen (the screen is cleared when ACCEPT begins execution) and then receive and validate data entered by the operator in response to this display. Current values of receivers in an ACCEPT statement are displayed, and may be altered by the user. ACCEPT can control positioning of data and literals on the screen, as well as format and display mode of data (bright, dim, flashing, etc.). Data entered to an ACCEPT statement can be automatically validated by type (alpha or numeric) and range of values; data not of the appropriate type or value is rejected and must be re-entered by the user. ACCEPT can also perform branches to other statements based on the use of PF keys and on whether or not displayed data values are altered. ACCEPT cannot branch to marked subroutines, as can INPUT. The ACCEPT statement is discussed in detail in Section 7.5, The ACCEPT Statement, and in Part 2.

7.2 PRINTER OUTPUT

Most printers have 132 columns, numbered left to right from column 1 through column 132. The columns are divided into seven zones; zones begin in columns 1, 19, 37, 55, 73, 91, and 109. All zones occupy 18 character positions, except the rightmost zone, which is 24 characters wide.

Data can be output to the printer by using the PRINT statement after a SELECT PRINTER statement has been executed. Either literals or the current value of any variable or expression may be output, using a wide variety of formats.

The PRINT statement actually moves data to a line buffer for the printer. The contents of the line buffer is printed only when an implied or explicit move to the next line occurs (e.g., via the SKIP clause of a PRINT statement or via a PRINT statement with no trailing semicolon) or when data overflows the capacity of the line buffer. When the contents of the buffer has been printed, it is cleared and restarted at the first position.

The BASIC program may conclude a print operation by printing the contents of the line buffer with or without advancing to the next line (line feed). No line feed allows a program to overprint one line with another; line feeds are suppressed by ending a PRINT statement with a semicolon. See Part 2 for details. The program may also cause an arbitrary number of blank lines to be fed from the printer (with the SKIP clause of the PRINT statement).

Normally, if the BASIC program outputs too many characters to fit on the current print line, as many characters as possible are placed in the line buffer, the contents of the buffer are printed, and the remaining characters are moved to the start of the buffer for printing on the next line. This is equivalent to the "wraparound" phenomenon in workstation output (see Subsection 7.3.1, Wraparound).

For details on the use of the PRINT statement, see Part 2.

7.2.1 Expanded Print

Wang VS BASIC also provides expanded print capabilities. When a printer has been SELECTed, double width letters may be printed on a line-by-line basis. The command PRINT HEX(OE) as the first character of a line will initiate the expanded print, which will continue until a carriage return is encountered. The maximum number of expanded print characters that will fit on a line is 61. The carriage return automatically cancels the expanded print option. If multiple lines of expanded print are desired, each line must begin with the PRINT HEX (OE) command.

7.3 WORKSTATION INPUT/OUTPUT

The workstation display contains 24 rows of 80 characters, for a total of 1920 character positions. Each character position in the display can be referred to by its row and column number. Thus, position (1,1) is the first position on the top row; position (24,1) is the first position on the bottom row. All the positions in a row form a line. The PRINT statement further divides each line into zones which begin at columns 1, 19, 37, and 55 (these correspond to the zones used in printer output).

7.3.1 Wraparound

The entire workstation screen can be thought of as one sequential record containing 1920 bytes (actually, the record contains 1924 bytes, of which the first 4 are control characters normally transparent to the user). The order of bytes is from left to right within each line, and from each line to the one below. Thus, a character position to the right of another position on the same line is thought of as being "beyond" the position to its left. Similarly, a character position on a physically lower line of the screen is "beyond" a character position on a physically higher line. Each line is considered to "wrap around" to the next line: column 1 of any line is thought of as directly following column 80 of the line above it. Thus, if a string of characters is directed to be output to a line on which there is not enough space remaining to fit the specified characters, as many as possible will be displayed on the current line, and the rest will be displayed on the next line down. However, that column 80 of line 24 (the "end of the screen") does not wrap around to column 1 of line 1.

7.3.2 Scrolling

If wraparound occurs when the cursor is at the end of the screen, or if the cursor is explicitly directed to move down one line when already on the bottom line of the screen, all data then displayed on the screen is shifted up one line, so that the cursor appears to move down relative to the text on the screen. This operation is called an "upward scroll" or "roll-up." In like manner, a command to move the cursor up past the top line of the screen will result in all the text displayed on the screen shifting down one line -- a "downward scroll" or "roll-down."

In a scroll, a new line filled with spaces (ASCII code HEX(20)) appears on the screen, and one line leaves the screen. The data on the line which leaves the screen is not recoverable by the program.

7.3.3 Field Attribute Characters (FACs)

Any position on the screen may contain any 8-bit (1 byte) binary code. The codes from HEX(00) to HEX(7F) represent characters which can be displayed on the workstation screen. HEX(20) is the "space" or "blank" character. HEX(00) also displays as a blank.

The codes from HEX(80) to HEX(FF) are Field Attribute Characters (FACs). FACs also occupy character positions, but do not display a graphic character. FACs define the start of a field and contain information which will be applied to all character positions beyond it until either another FAC occurs or the end of the line is reached. This information governs the following decisions:

1. Whether the field will be displayed <u>bright</u>, <u>dim</u>, <u>blinking</u>, or <u>nondisplay</u> (i.e., displayable characters of the field will be suppressed). These four options are mutually exclusive.

2. Whether an <u>underline</u> will appear in all character positions in that field, or in none.

3. Whether or not the field is <u>modifiable</u> by operator input (<u>protected</u>).

4. Whether (a) no restrictions are placed on operator input, (b) lowercase letters input will be capitalized, or (c) only digits 0 through 9, decimal point, and minus sign will be allowed as input. Note that this affects only <u>input</u>; any characters can be output in any field type. This information is irrelevant if the field has already been declared "protected" by Option 2.

Appendix G contains a list of the Field Attribute Characters.

When BASIC programs are running, the conditions assumed at the start of each line are: (1) dim display, (2) not underlined, and (3) protected. There is an "assumed" FAC (HEX(8C)) with those characteristics to the immediate left of column 1 of each line.

The programmer may output FACs at any time by specifying the correct hexadecimal code in any screen I/O statement. For example,

```
300 PRINT HEX(94);
```

places on the screen at the current cursor position a FAC which causes data displayed to its right to be blinking, with no underlining, and protected.

The INPUT statement places a FAC (of HEX(81)) in the screen buffer to the left of the field where input is to occur, thus setting that field to "bright, no-line, modifiable, uppercase."

The ACCEPT statement causes a FAC to be placed before each input field. This FAC will normally specify (1) bright display, (2) not underlined, and (3) modifiable. The setting of Option 4 depends on the type of item to be input in that field. If a string is to be input, the setting will be "no restrictions on input" (HEX(80)). If a floating-point number is to be input, the setting will be "uppercase only" (HEX(81)), to allow input of +, -, ., and E. If an integer is to be input, the setting will be "numeric only" (HEX(82)). The programmer may override these FAC values by making an explicit specification with a FAC clause in the ACCEPT statement. (See Subsection 7.5.1, Screen Formatting.)

Unless the input field is followed immediately by another input field, the ACCEPT statement places an additional FAC (HEX(8C)) at the end of the field to revert the display to the default settings.

7.4 THE USING CLAUSE AND FORMAT CONTROL STATEMENTS

The PRINT statement and a number of file I/O statements (see Section 8.4, The File I/O Statements) can use an auxiliary statement to define the format of data to be output or input. This format-control option is specified by including a USING clause, which contains the line number or statement label of either a FMT statement or an Image (%) statement. For example:

```
15600 PRINT USING RADISH, list of expressions
   .
   .
   .
33200 RADISH:  FMT list of format specifications

67200 %Output image
   .
   .
   .
73600 PRINT USING 67200, list of expressions and/or literals
```

Note that the position in the program of the FMT or % statement relative to the statement containing the USING clause is irrelevant.

The FMT and % (Image) statements are nonexecutable statements which contain formatting information for an I/O statement containing a USING clause.

7.4.1 The FMT Statement

A FMT statement consists of the reserved word FMT, followed by a list of control specifications, data specifications, and literals. Control specifications are clauses that determine the placement of data; they specify tab stops, column positions, and numbers of spaces or lines to be skipped. Data specifications are clauses that determine the type and format of particular data values to be input or output: alpha or numeric, number of digits to each side of decimal point, retention or suppression of leading zeroes, etc. For the general form of the FMT statement and a list of the kinds of control and data specifications, see Part 2.

Example:

```
FMT COL(10), CH(8), XX(2), PIC(####.##)
```

The control and data specifications in this statement are:

COL(10)	Control specification indicating that the first data item begins at the tenth position on the workstation or printer line (or, if used for file I/O, the tenth byte of the record).
CH(8)	Data specification for an alphanumeric ("CHaracter") value 8 characters long.
XX(2)	Control specification indicating that two spaces are to be skipped.
PIC(####.##)	Data specification giving an "image" or "picture" of a numeric value with four digits to the left of the decimal point, two to the right.

7.4.2 The Image (%) Statement

An Image (%) statement consists of the single character "%" followed by an image or "picture" of how the output data will look. Fields of number signs (#) act as data specifications, which show where and how data values will be input or output. Unlike in the FMT statement, there are no control specifications; the information which would be given by control specifications in a FMT statement is given in an Image (%) statement by the actual layout of the fields of number signs. Special editing characters in these fields indicate the placement of signs, decimal points, commas, exponent fields and other special characters used with numeric data. Fields which describe separate data items must be separated by one or more spaces. For a detailed description of the Image (%) statement, see Part 2.

Example:

```
%   ###  UNITS @  $####.##
```

This statement has two data specification fields and a literal. The first data specification field is three characters long, beginning at the fourth character position of the workstation or printer line (fourth byte of a record if used for file I/O). This is followed by the literal "UNITS @" and the second data specification field, eight characters long, beginning at the 18th character position (byte). Either alpha or numeric data could be input or output through either of these data specification fields. Numeric data output through the second would appear with two digits to the right of the decimal point, and a dollar sign to the left of the leftmost digit.

7.4.3 Use of FMT and Image (%) Statements

PRINT and file I/O statements with USING clauses may contain a list of expressions which are to be output. Starting at the beginning of the list, items from the list of the PRINT or file I/O statement must correspond with the data specifications in the FMT or Image statement. Thus, to print a numeric value followed by an alpha value through an FMT statement, the FMT statement must contain a numeric data specification followed by an alpha data specification. For example:

```
1400 PRINT USING JUVENESCENCE, MAGNITUDE, NAME$
1500 JUVENESCENCE:  FMT PIC(####), CH(16)
```

prints the current value of MAGNITUDE using the specification PIC(####), and then prints the current value of NAME$ using the specification CH(16). Any attempt to input or output data through an FMT or Image (%) statement whose data specifications do not match those of the data actually presented will result in a data conversion error at run time. If the error is caused by a PRINT statement, execution halts and an error message is displayed. If caused by a file I/O statement, the branch indicated in the data error exit clause (see Section 8.6, Error Recovery) is taken; if no data error exit was specified, execution halts with an appropriate error message.

If there are more items in the PRINT or file I/O statement than there are data specifications in the FMT or Image (%) statement, the FMT or Image (%) statement is reused, as though it were replicated as many times as necessary to accommodate the remaining items in the list of the I/O statement. An error message will be produced if a PRINT or file I/O statement with a non-null argument list is used in conjunction with a FMT or Image (%) statement containing no data specifications. In PRINT USING, subsequent output will occur on the next line down unless the item in the PRINT USING statement which exhausted the FMT or Image (%) statement was followed by a semicolon.

If there are more data specifications in the FMT or Image (%) statement than there are items in the PRINT or file I/O statement, the remainder of the FMT or Image statement is ignored. The I/O operation ends at the first data specification without a matching item from the I/O statement. This situation can also occur when an FMT or Image statement is reused, but contains more data specifications than there are items remaining in the I/O statement.

For example,

```
1100 %-### XYZ -###.##
1400 PRINT USING 1100, E, F, G
```

will display the current contents of E using the Image "-###", then the literal "XYZ", and then the current contents of F, using the image "-###.##". Now G remains in the PRINT USING list, but the Image (%) statement is exhausted. Therefore, the process is begun again, and since F and G are separated by a comma instead of a semicolon, subsequent display will occur on the next line down. G is output using the image "-###" and XYZ is printed on the same line. Printing stops here, since there are no more arguments to use the next data specification.

7.5 THE ACCEPT STATEMENT

ACCEPT is the most versatile of the VS workstation I/O commands. A single ACCEPT statement displays a formatted screen of data at the workstation, which may include literal messages as well as the values of numeric and alpha expressions. New values can be entered by the user for receivers displayed on the screen. Note that ACCEPT processes an entire screen full of data at one time, instead of one line at a time, as is done by INPUT. An ACCEPT statement consists of the word ACCEPT, followed by a list of items to be displayed on the workstation screen. A single ACCEPT statement may perform any or all of the following functions, depending on the use of various optional clauses:

1. Output literal messages.

2. Display current values of variables and alpha receivers in an optionally specified format (PIC, CH clauses).

3. Control placement of displayed literals and receivers (AT clause).

4. Control display mode of displayed receivers (FAC clause).

5. Accept modifications to the values of displayed receivers.

6. Validate modifications to receivers to confirm that they are within a specified range of values (RANGE clause).

7. Accept input from Program Function Keys (KEYS, KEY clauses).

8. Branch to different places in a program depending upon the user's response to the ACCEPT statement (ON and NOALT clauses).

For the general form of the ACCEPT statement, see Part 2. The following sections illustrate the use of each of the optional clauses by example.

7.5.1 Screen Formatting

Fields

When an ACCEPT statement is executed, the entire screen is cleared and a screen is displayed containing items specified in the ACCEPT statement. The items that can be displayed are literal messages, numeric variables, and alpha receivers. Unless the programmer specifies otherwise new values may be entered for variables and alpha receivers by typing over the displayed values. Displayed literals, however, cannot be modified.

Each item that is displayed occupies a field on the CRT screen. A field is a sequence of adjacent character positions on the workstation screen which is associated with a particular item in an ACCEPT statement. The width of each field is equal to the number of characters required to display the item. In the case of literals, this is simply the length of the literal itself. For alpha receivers, the defined length is used as the field width unless some other width is specified with a CH clause. Field width for numeric variables is 18 characters unless some other width is specified with a PIC clause.

ACCEPT explicitly shows the field that a variable or alpha receiver occupies by displaying all blank spaces in the field as pseudoblanks. These appear on the screen as solid squares, and will be shown in the following examples as underscores. For example:

```
100 A=99
200 B$="BOTTLES"
300 ACCEPT A, B$, "OF BEER ON THE WALL."
400 PRINT A, B$
```

will generate a screen containing the line

```
99_______________ BOTTLES_________ OF BEER ON THE WALL.
```

The value of A is output in a field 18 characters wide: a leading pseudoblank is shown where a minus sign would be displayed if the value were negative, followed by the digits 99, followed by 15 trailing pseudoblanks. Since B$ was not explicitly dimensioned, it has a default length of 16 characters. Thus, nine pseudoblanks are shown following BOTTLES. The literal OF BEER ON THE WALL is displayed exactly as written, with no pseudoblanks. Note that a space is displayed before the beginning of each field. This space contains a Field Attribute Character (see Subsection 7.3.4, Field Attribute Characters (FACs)).

Positioning Data on the Screen: The AT Clause

The AT clause can be used to position a field anywhere on the screen by specifying the row and column of the screen at which a particular field will begin. In the example above, the first value was displayed starting at the second column of the first row; the first column of every row is inaccessible for displaying data, since it always contains a FAC. If line 300 was changed to

```
300 ACCEPT AT (12, 25), A, B$, AT (13, 25), "BEER ON THE WALL."
```

the following would appear in the center of the screen (starting at the 25th column of rows 12 and 13):

```
_99______________ BOTTLES_________
OF BEER ON THE WALL.
```

If no AT clause is specified for a field, the field is positioned according to the default rules specified in Part 2.

Controlling Display Attributes for a Field: The FAC Clause

In addition to controlling the position on the screen of literal and receiver fields, the ACCEPT statement allows the programmer to specify the display attributes of a field. The display attributes determine whether a field will be shown as dim, bright, blinking, or non-displayed; modifiable or protected; containing uppercase, numeric, or all characters; underlined or not underlined. Display attributes are controlled by displaying a Field Attribute Character (a FAC; see Subsection 7.3.4, Field Attribute Characters (FACs)) immediately preceding a field. For example, in the example above, changing line 300 to

```
300 ACCEPT AT (12,25), A, FAC(HEX(91)), B$, AT (13,25),           !
350   "OF BEER ON THE WALL."
```

would cause the same message as before to be displayed, except that the B$ field (containing the string BOTTLES) would be blinking, since HEX(91) is the FAC specifying blink, modify, uppercase, no line.

Note that FACs cannot be specified for literal fields, which are always shown preceded by a FAC of HEX(AC) (dim, protect, all, no line).

If the display attributes for a particular field are not explicitly defined with a FAC clause, the following defaults are used:

Alphanumeric	-	bright, modifiable, uppercase, no underline (HEX(81)).
Floating-point	-	bright, modifiable, uppercase, no underline (HEX(81)).
Integer	-	bright modifiable, numeric only, no underline (HEX(82)).

Format Images of Displayed Receivers: The PIC and CH Clauses

It is often useful to be able to display modifiable data fields in formats other than the default formats described above. This can be done with the PIC clause for numeric fields and with the CH clause for alpha fields. Each of these optional clauses appears directly after the receiver it modifies in the ACCEPT statement, separated from the receiver by a comma. The PIC clause specifies a format image for numeric data, and the

CH clause specifies a field width in characters. For a description of all of the editing characters which may be used in a PIC clause, see the discussion of this clause under the FMT statement in Part 2. For example, if it was known that the variable A would never require more than three character positions to be displayed,

```
300 ACCEPT AT (12, 20), A, PIC(###), B$, CH(7), "OF BEER ON THE WALL."
```

This would display

```
_99 BOTTLES OF BEER ON THE WALL.
```

on line 12 of the workstation screen. Note that the A and B$ fields no longer contain trailing pseudoblanks.

7.5.2 Data Entry and Validation

When an ACCEPT screen is first displayed, the cursor is positioned at the first character of the first modifiable field. New values for any numeric variables or alpha receivers can be entered by typing over the displayed values in the appropriate fields, provided that the field has not had a PROTECT FAC placed before it.

The cursor may be moved to different fields on the screen by use of the cursor control keys (the four keys on the workstation marked with arrows) and by the TAB, BACK TAB, and NEW LINE keys. TAB moves the cursor to the beginning of the next modifiable field, BACK TAB moves it to the beginning of the previous modifiable field, and NEW LINE moves it to the beginning of the next modifiable field which is not on the current line. All three of these keys move the cursor without affecting any of the values displayed on the screen. A field may be set to all blanks from the current cursor position to the end of the field by pressing the ERASE key.

Any attempt to type over a non-modifiable field or to otherwise type characters prohibited by the FAC governing a particular field will cause the workstation alarm (a beep) to sound, and the cursor will not move.

None of the changes made to data on the workstation screen are actually transmitted from the workstation to the computer until the ENTER key (or a PF key) is pressed, allowing the user to modify data repeatedly until this point. When the ENTER key is pressed, the data displayed on the screen is transmitted from the workstation to the computer; execution then continues either with the next statement or with optional ACCEPT clauses.

Data validation: The RANGE clause

The optional RANGE clause may be used to perform automatic data validation to insure that data entered from the workstation falls within a specified range of values. A RANGE clause is inserted after the name of a receiver in an ACCEPT statement, separated from the receiver name by a comma, and applies only to that receiver. For numeric data, the range can be specified as being positive (RANGE(POS)), negative (RANGE(NEG)), or between the values of two expressions evaluated at run time (RANGE(exp1, exp2)). For alpha receivers, the range can be specified as being between the values of two alpha expressions in the ASCII collating sequence (RANGE(alpha-expl, alpha-exp2); see Subsection 5.2.3, Relational Operators, for a discussion of the ASCII collating sequence).

When ENTER is pressed during the execution of an ACCEPT statement, the value shown on the screen for each modifiable field is compared to the corresponding RANGE specification, if one exists. If any modifiable field contains a value which falls outside that specified, the screen is redisplayed with the first incorrect field blinking, and the user must re-enter the value. When all fields satisfy their RANGE specifications, execution continues. For example,

```
300 ACCEPT AT (12, 15), A, PIC(###), RANGE(50,100), B$, CH(7),       !
350   RANGE("BARRELS", "KEGS"), "OF BEER ON THE WALL."
```

will display

```
_99 BOTTLES OF BEER ON THE WALL.
```

If the value in the numeric field is changed to 45, the screen is re-displayed with that field ("45") blinking, since 45 is not within the specified range. Altering the value to any value between 50 and 100 will cause it to be accepted. Similarly, if BOTTLES is changed to LITERS, that field will flash when entered, since LITERS is not between BARRELS and KEGS in the collating sequence. CASES, however, will be accepted.

7.5.3 PF Key Usage and Program Branching

The ACCEPT statement allows the program to respond to specified Program Function (PF) keys. When a PF key is pressed, its value can be assigned to a variable for subsequent testing and branching (or for use in a numeric expression), or it can be automatically tested by ACCEPT, initiating an immediate branch to another statement (ON key clause). This capability is useful for writing interactive programs in which the user can select options or issue commands from a menu (see Subsection 1.2.2, Use of PF Keys: Menus).

If any of the three PF key clauses are present, they must appear after all of the literals, receivers, and modifying clauses. If more than one of these three appear, they must appear in the order in which they are discussed in the following paragraphs.

The KEYS Clause

The KEYS clause specifies which of the 32 PF keys will be processed by an ACCEPT statement. Pressing any PF key which has not been enabled by a KEYS clause causes the workstation alarm to beep, and the key is ignored. KEYS must be followed by an alpha-expression in parentheses, which is interpreted as a list of one-byte binary values corresponding to the numbers of the PF keys to be accepted. Such a list can be specified either with the BIN function or with the HEX function. In the latter case, the PF key numbers must be converted to hexadecimal.

For example, to enable PF keys 1, 12, and 15, one would write either

```
KEYS(BIN(1) & BIN(12) & BIN(15))
```

or

```
KEYS(HEX(010C0F))
```

Once any valid PF key is pressed, execution of the program continues and data on the screen can no longer be modified.

The KEY Clause

The KEY clause assigns the value of whatever valid PF key is pressed to a numeric variable, which is specified in parentheses after the word KEY. For example, if the clause KEY(OPTION) is in an ACCEPT statement, and the user presses PF 15, the value 15 is assigned to the variable OPTION, and execution continues.

The ON Key Clause

The ON key clause enables the program to branch to different points depending upon which PF key is pressed. The ON key clause can perform either GOTO or GOSUB branches. In either case, the branch is taken without reading any changes made to the screen. As in the KEYS clause, ON is followed by an alpha expression which is treated as a list of 1-byte binary values of PF key numbers. The GOTO or GOSUB verb must be followed by a list of line numbers and/or statement labels to which branches may be made. The position of each line number or statement label in this list must correspond to the position of its associated PF key in the list of PF key numbers following the word ON. Thus, pressing the first PF key listed would initiate a branch to the first line number or statement label, the second PF key to the second line number or label, and so on. For example,

```
ON (BIN(1) & BIN(16)) GOTO 100, FINISH
ON (BIN(5) & BIN(9)) GOSUB CATERPILLAR, BUTTERFLY
```

would cause control to be transferred to line 100 if PF 1 is pressed, to the statement labeled FINISH if PF 16 is pressed, and to the appropriate subroutines if PF 5 or 9 is pressed.

The ALT and NOALT Clauses

When data entry to an ACCEPT statement is terminated by ENTER or a PF key, the ACCEPT statement can automatically determine if any field has been modified. Any keyboard action that is performed on a field, even retyping the old data or erasing pseudoblanks, will indicate that the field has been altered.

The ALT clause can be used to increase the efficiency and speed of screen processing by causing the ACCEPT statement to read, validate, and transfer only those fields which were actually modified.

The NOALT clause is a conditional branch clause which can perform either GOTO or a GOSUB branch to a line number or a statement label. If this clause is included, and none of the displayed fields are altered, control is passed to the specified line or statement. If any fields are altered, only those which have been altered are read, validated, and transferred (as with ALT), and execution continues without taking the specified branch. For example,

```
NOALT GOTO LILLIPUT
NOALT GOSUB 37200
```

ALT and NOALT cannot both appear in a single ACCEPT. This would be redundant, anyway, since NOALT performs the function of ALT if any fields are modified.

7.5.4 Summary of ACCEPT Execution

1. The screen is generated as described, with the cursor positioned at the first modifiable (or numeric-protected) field, if any are present. All fields contain the current values of the receivers.

2. The user may enter new values. When ENTER is keyed, or a PF key is pressed, the key is first checked for validity. If invalid, the workstation emits a beep, and the user may continue modifying or press another key.

3. If the key is specified in the ON clause, the specified branch is taken without any field reads or verification. (The KEY variable, if specified, will contain the key number in any case.)

4. Otherwise, all modifiable fields (or only altered fields if ALT or NOALT is specified) are read/validated. Numeric fields are validated for proper numeric format independently of range validation. Although any PIC specification may be used, special characters (CR,DB, etc.) are <u>not</u> valid on input.

   If any field is invalid, its FAC is set to blinking and the user must correct the mistake (and can further change other fields).

Example:

```
300 ACCEPT AT (12, 15), A, PIC(###), RANGE(50,100),                  !
310   FAC(HEX(91)), B$, CH(7), RANGE("BARRELS", "KEGS"),             !
320   "OF BEER ON THE WALL.",                                        !
330   KEYS(BIN(0) & BIN(1) & BIN(16)), KEY(OPTION),                  !
340   ON (BIN(1) & BIN(16)) GOTO START, FINISH,                      !
350   NOALT GOSUB 1700
```

7.6 THE DISPLAY STATEMENT

DISPLAY, like ACCEPT, clears the workstation screen and displays an entire formatted screen at one time. Unlike ACCEPT, however, DISPLAY does not accept any input either of data values or PF keys.

DISPLAY can position data with the AT clause, and specify formats of displayed numeric and alpha data with the PIC and CH clauses. These three clauses all operate as in ACCEPT. DISPLAY can also position data with the COL clause, which has the form COL(n), where n is some integer. COL(n) specifies that the next data item is to be displayed starting at the n-th column of whichever row the cursor is currently on.

For further details on DISPLAY, see the entries on DISPLAY and ACCEPT in Part 2.

7.7 WORKSTATION PROGRAMMING CONSIDERATIONS

When programming output to the VS workstation, it is important to keep in mind that the workstation is capable of producing output very much faster than a human user can read. Incautious use of statements which clear the workstation screen may lead to output being erased from the screen before it can be read by the user. For example, if two DISPLAY statements follow one another with few or no intervening statements, the data displayed by the first DISPLAY may be on the screen for only a fraction of a second before it is erased by the second DISPLAY statement. (The actual duration of the screen display will depend on many variables at execution time, including how many other users are logged onto the VS, how much main memory is available to each user, etc.)

There are several ways to avoid this problem. The SELECT P[d] statement can be used to make the program pause for d/10 seconds after each DISPLAY or PRINT to the workstation. The pause interval remains the same until another SELECT P[d] is encountered, irrespective of the amount of data displayed (and therefore irrespective of the time required to read the screen).

The STOP statement can be used after a DISPLAY or PRINT statement to halt execution while the user reads a screen. Execution is resumed when the user presses the ENTER key or a legal PF key (one which corresponds to a marked subroutine; see Subsection 6.4.3, Program Function Keys). The STOP statement displays the word STOP and an optional literal when it is executed.

Since all of the functions of the DISPLAY statement (except sounding the workstation alarm) can be performed by the ACCEPT statement, ACCEPT statements can be used to display information without performing any meaningful input. To allow the user time to read a long message on the screen, one can display the message as a series of literals and/or expressions with the ACCEPT statement, and enable only the ENTER key (PF 0, specified with a KEYS(BIN(0)) clause) for input. When the ACCEPT statement is executed, the desired message will be displayed on the screen, and will remain there until the user presses the ENTER key.

CHAPTER 8
FILE INPUT/OUTPUT

8.1 INTRODUCTION

Programs written in VS BASIC can retrieve and store data in files located on magnetic disk or tape. The first part of this chapter (Section 8.2, Files) provides general background information on the types of files supported on the VS, and their attributes and structures. Only that information required for the use of the BASIC file input/output facilities is included. Programmers requiring further detail on the organization of VS files and on the operation of the Data Management System should consult the VS Programmer's Introduction and the VS Operating System Services manuals. The rest of this chapter describes the specific features of VS BASIC that facilitate file I/O operations, including detailed examples of the use of the most frequently used file I/O statements.

8.2 FILES

A file is a collection of data stored on either magnetic disk or tape, and identified by a file name. Files are made up of records. A record is the unit of all file input/output operations, and consists of a continuous series of bytes of data which are processed together. In general, a record corresponds to whatever unit of data is logically most convenient to process at one time. For example, in an inventory control program, a single record might consist of a part number, the quantity in stock, quantity on order, order date, price, and so on. An inventory file for 100 different parts would thus contain 100 records. In a file of text (for example, a BASIC source file), each record would correspond to a line of text as shown on the workstation or printer.

8.2.1 File Types

The VS supports three different types of disk files -- consecutive, indexed, and print files -- which differ in their internal organization and use. The details of internal file organization are transparent to the BASIC programmer, since these are all managed and maintained by the Data Management System. This section contains a general discussion of the three file types. Section 8.3, Use of Files by BASIC Programs, describes the BASIC statements which control file selection and use.

Consecutive Files

A consecutive file contains records which, within a block (a block is 2048 (2K) contiguous bytes of storage space), physically follow one another in the same order in which they were written. The information in a record does not in any way influence its position within the file. The position of a record relative to other records in a file therefore depends only on when it was written relative to other records.

Records in a consecutive file may be read either sequentially or by position. In sequential reading the user program in effect says to the Data Management System (DMS; see Subsection 1.3.1, The Data Management System) "get the next record," whereas reading by position is equivalent to saying "get the *n*-th record."

Indexed Files

An indexed file is a disk file that contains records in a logical sequence which is not necessarily the same as the order in which the records were written (as would be true of a consecutive file). The logical sequence of records in an indexed file is determined by the value of the *primary key* of each record. A primary key is a designated portion of a record (the eighth through the twelfth bytes of each record, for example) that is used to sort the records into a particular ordered sequence. Each record in an indexed file must have a unique primary key.

Indexed files may also have *alternate keys*. Like the primary key, an alternate key is a section (or field) of a data record of some designated position and length. One file may use up to 16 alternate keys, numbered 1 to 16. Every record in an indexed file has associated with it an alternate index *mask*, which is a two-byte (16-bit) field specifing by which alternate keys that record may be accessed.

Indexed files enable a program to access particular records according to primary or alternate key value. Records may be read from an indexed file sequentially or by key. In sequential reading, the program in effect says to DMS, "get the next record in ascending primary key sequence." Reading by key, on the other hand, is equivalent to saying "get the record with key equal to *x*."

Indexed files are organized into *data blocks* and *index blocks* (a block is 2048 (2K) contiguous bytes of storage space). Data blocks contain the actual data records, in primary key sequence, within each block. The index blocks contain a list of primary key values of records in the file, with a corresponding list of pointers that indicate in which block a record with a particular key can be found. Files with alternate keys have a separate numbered alternate index for each defined alternate key field. Each alternate index contains entries only for those records whose alternate index masks specify that they are accessible by that key.

The first time an indexed file is opened for output (i.e., when it is created), any records written to it must be written in key sequence. Later additions to the file may be made in any order. DMS takes care of inserting the data records and index entries into their respective blocks at the appropriate points. When either an index block or a data block becomes full, it is split -- the contents of half of the block are moved to an empty block. The result is two half-empty blocks, instead of one full block. New insertions can then be made in the two blocks until one or both are full, and the splitting process takes place again.

Print Files

A print file is a disk file used to store records which are to be output by a printer. The records in a print file, like those in a consecutive file, always appear, and will be printed, in the order they were written. In addition to the data records, print files contain printer control bytes which contain information affecting the physical appearance of print on a page (line-feed codes, page breaks, etc.). Print files can be written but not read by BASIC programs. Print files are generally only read by printer Input/Output Processors (IOPs) and certain system utility programs (e.g., DISPLAY).

8.2.2 Record Types: Length and Compression

Record Length

Files may contain records which are all of the same length (fixed-length records) or of differing lengths (variable-length records). Files containing fixed-length records have the record length specified in the file label. In files with variable-length records, the length of each record is specified by a length count at the beginning of each record (which is maintained by DMS and is thus transparent to the user).

Record Compression

Variable-length records may also be compressed. If record compression is specified for a file, characters which are repeated three or more times consecutively are stored on the disk only once, preceded by a repetition factor. Compression is performed automatically by DMS when information is moved to the disk; compressed records are decompressed when the reverse transfer is performed. The entire compression/decompression process is completely transparent to user programs. Record compression can often save substantial amounts of disk space.

8.3 USE OF FILES BY BASIC PROGRAMS

All transfers of data between user programs and files are processed by the Data Management System (see Subsection 1.3.1, The Data Management System (DMS)). The user program communicates with DMS about the files to be used through User File Blocks (UFBs). A UFB contains information about fixed characteristics of the file it describes: whether it is a consecutive, indexed, or print file; the record type (fixed- or variable-length, compressed or not); record length; and various other factors. When a BASIC program is compiled, one UFB must be created for each file of particular characteristics to be used by the program. These characteristics are all specified in a SELECT statement, which also assigns a file-number to a UFB. Note that since the UFB is part of the object program, a SELECT statement has its effect (creation of a UFB) at compile time.

In addition to the fixed characteristics of a file specified with SELECT, there are factors relating to the way in which a file is to be used which are specified at run time and which may change during the execution of a program. These include the file's name, whether it is to be used for input or output, and how much space is allocated for it if it is a new file. These run-time specifications are made with the OPEN statement. The OPEN statement initiates a connection between the user program and a specific file through a particular UFB by associating the name of the file with the file-number of the UFB. This connection is severed by the CLOSE statement. I/O operations can be performed with a file only if it is open. The characteristics (file type, record type and length, etc.) of a file named in an OPEN statement must match those specified in the SELECT statement for the file-number if the file already exists. If the file is being created, it is created with the characteristics described by the SELECT statement.

Only one file can be opened on a particular file-number at a time. Thus, a program must contain one UFB, and thus one SELECT statement, for each file with a particular set of characteristics to be open at one time. For example, a program might use one consecutive file, one indexed file, and one printer file. In this case, it must have three SELECT statements, one to create each of the three different UFBs needed. Another program may use three consecutive files, each with fixed-length, 80-byte records. If no more than one of these files is open at one time, then the program needs only one SELECT statement. All three files can use the same UFB, since they will be open at different times.

8.3.1 The SELECT Statement

The SELECT statement is made up of a series of clauses, some of which are optional, which describe the characteristics of the file(s) that may be associated with a particular UFB. For the general form of the SELECT statement, see Part 2.

The elements of the SELECT statement indicate the following:

File-number -- Number sign (#) followed by an integer from 1 to 64 (inclusive). This file number is used in all other I/O statements to refer to the file described by this SELECT statement. Must be specified for each UFB to be created.

Prname -- A literal string consisting of 1-8 alphabetic or numeric characters. Must be specified for each UFB created. The prname is not the filename.

VAR[C] -- Specifies that records are variable length (optionally compressed). If not specified, records are fixed length, with length specified by the RECSIZE clause. Cannot be specified for PRINTER files; optional for all other types.

CONSEC, INDEXED, PRINTER, TAPE -- Specifies file type. These are mutually exclusive alternatives; one of the four types must be specified. Note that a TAPE file is always consecutive in form. The word CONSEC always indicates a consecutive disk file, however.

RECSIZE=int1 -- Record length, in bytes, for files with fixed-length records. Maximum record length for files with variable-length records. Must be specified for every file, irrespective of type. See SELECT entry in Part 2 for record length limits for each file type.

KEYPOS=int2 -- Position (starting from byte number 1) of first byte of primary key in records of an indexed file. Must be specified for an indexed file.

KEYLEN=int3 -- Length (in bytes) of primary key in records of an indexed file. Must be specified for an indexed file.

ALT[ERNATE] KEY int4, KEYPOS=int5, KEYLEN=int6 -- Number, position, and length of an alternate key in the records of an indexed file. Optional for an indexed file. Up to 16 alternate keys may be specified; each must be identified by a unique key number (int4). ALT and ALTERNATE are equivalent forms.

DUP -- Indicates that duplicate key values are allowed for the alternate key specified in the preceding clause. For indexed files only. If not specified for an alternate key, any duplicate value found for that alternate key will cause the EOD exit to be taken.

IL, NL, AL -- Specifies the tape label type for TAPE files only. IL = IBM-type label, NL = no label, AL = ANSI standard label.

BLKSIZE -- Specifies the size, in bytes, of the blocks into which TAPE files are divided.

IOERR -- Specifies a GOTO or GOSUB branch to be taken if an I/O error occurs on the file which is OPENed on the SELECTed file number (see Subsection 8.3.2, The OPEN and CLOSE Statements). Optional for all file types.

EOD -- Specifies a GOTO or GOSUB branch to be taken if an end-of-data condition, invalid key or duplicate key is found in a file while performing an I/O operation which does not have an EOD exit of its own. Cannot be specified for PRINTER files; optional for all others (see also Section 8.6, Error Recovery).

NOTE:

The prname (parameter reference name) specified in the SELECT statement is not a file name. The actual name of the file to be used is specified in the OPEN statement. The prname is used by DMS to refer to a file at run time in requests for information (called GETPARMS) displayed on the workstation screen. Such requests appear when DMS requires information not specified (or incorrectly specified) in the program. GETPARMs are discussed more fully in Subsection 8.3.2, The OPEN and CLOSE Statements.

Note that one SELECT statement must be written for every UFB to be used by a program. All SELECT statements must appear in a program before any OPEN or file I/O statements.

File numbers need not be SELECTed consecutively. No file number may be used in more than one SELECT statement.

8.3.2 The OPEN and CLOSE Statements

The OPEN statement enables input or output between a BASIC program and a file, and associates the file name with the file-number of a particular User File Block which has already been created by a SELECT statement. In all subsequent file I/O statements, the file is referenced by the file-number in the OPEN statement.

The CLOSE statement is used to terminate the connection between a file and a numbered User File Block. If a file is open through a particular UFB, no other file can be opened through that UFB until the first file is closed. The form of the CLOSE statement is: CLOSE #file-expression.

The name of the file may be specified in the OPEN statement or the user may be prompted for it when the OPEN is executed. In the latter case, a GETPARM is issued. A GETPARM is a request issued by the Data Management System for information needed to perform certain operations. When a program is being run directly from a workstation, a GETPARM displays a screen specifying what information is needed; the user types this information into the appropriate fields on the screen, presses the ENTER key, and execution continues. If the program is being run from a procedure, the procedure is first prompted for the information; the GETPARM screen is displayed only if the procedure does not supply all of the necessary information, or if some of the information is in error (see the VS Procedure Language Reference Manual for a discussion of procedures).

If the file, library, and volume names are specified in the OPEN statement and need not be changed, the GETPARM prompt can be suppressed by specifying NOGETPARM in the OPEN statement. The prompt screen may be suppressed by NODISPLAY. This should be used only if the correct file, library, and volume names have been specified in this or an earlier OPEN, or in a procedure running the program, or if SET defaults are in use (see the VS Procedure Language Reference Manual for a discussion of procedures and SET defaults). The difference between NOGETPARM and NODISPLAY is that the former should be used only if the file, library, and volume names are specified in the current OPEN statement; NODISPLAY can be used if these specifications are omitted from the statement, as long as they can be obtained elsewhere (earlier OPEN statement, procedure, or SET defaults).

Even if the file, library, and volume names are specified in the OPEN statement, the GETPARM screen will be displayed if neither NOGETPARM nor NODISPLAY is present. In this case, the user can simply press ENTER, which causes the file specified in the OPEN statement to be used, or can alter the displayed file, library, and volume names.

See Part 2 for the general form of the OPEN statement.

The elements in the OPEN statement indicate the following:

NOGETPARM -- Suppresses the issuing of a GETPARM for the file, library, and volume names. Should be used only if these are specified in the OPEN statement. Optional and mutually exclusive with NODISPLAY.

NODISPLAY -- Suppresses display of a GETPARM screen for file, library, and volume names,. An "invisible" GETPARM is issued to the controlling procedure if one exists or else the required information is obtained from this or an earlier OPEN or from the SET defaults. Optional and mutually exclusive with NOGETPARM.

File-exp -- A numeric expression which is evaluated to obtain the file-number by which this file will be referenced by all file I/O statements.

INPUT -- Opens an existing file for input. The program will then be able to read from the file, but not modify it. Mutually exclusive with IO, OUTPUT, EXTEND, and SHARED. (Does not apply to print files.) See Subsection 8.3.3, File I/O Modes.

IO -- Opens an existing file for input and output. The program will then be able to read and modify the contents of the file. For an indexed file, I/O mode allows the addition of new records (like EXTEND for consecutive files). A consecutive file with fixed-length records may do REWRITES in I/O mode, but may not create new records. Mutually exclusive with INPUT, OUTPUT, EXTEND, and SHARED. (Does not apply to printer or tape files.) See Subsection 8.3.3, File I/O Modes.

OUTPUT -- Specifies that a new file is to be created and opened for output, in which case records may be written out to the file, but cannot be read from that file. If the file existed previous to the OPEN, the user will be asked to either delete the old file or specify a new name. Printer files can only be OPENed in this mode. Mutually exclusive with INPUT, IO, EXTEND, and SHARED. See Subsection 8.3.3, File I/O Modes.

EXTEND -- Opens an existing consecutive file for extension. The program will then be able to write to the file, but not read from it. The first record written will be stored directly following the last record which was already in the file. Mutually exclusive with INPUT, IO, OUTPUT, and SHARED. (Does not apply to printer files.) See Subsection 8.3.3, File I/O Modes.

SHARED -- Opens an existing file in shared mode. This mode is similar to I/O mode, but allows simultaneous access to the file by other VS users. Mutually exclusive with INPUT, IO, OUTPUT, and EXTEND. (Shared mode is supported only for indexed files and for a special type of consecutive file called a "log file.") See Subsection 8.3.3, File I/O Modes.

SPACE=expl -- For OUTPUT mode files, specifies the approximate number of records (expl) to be put in the new file. If SPACE is omitted, a GETPARM will be displayed to request the required space information. For non-OUTPUT mode files, if expl is a numeric variable, it will be assigned the number of records in the file when the file is opened.

DPACK=exp2, IPACK=exp3 -- For new (OUTPUT mode) indexed files only, specifies the packing densities of data and index blocks, respectively, in percent. The packing density determines what percentage of each data and index block will be filled with data records and index entries. This affects the efficiency of disk space use and the number of records which can be inserted into a file before DMS must reorganize a file by splitting data and/or index blocks (see discussion of indexed files, Subsection 8.2.2, File Types). For most efficient use of disk space, files which will never have additional records inserted should have DPACK and IPACK set equal to 100. Files which will have records inserted should set DPACK and IPACK to values less than 100.

FILE=alpha-expl, LIBRARY=alpha-exp2, VOLUME=alpha-exp3 -- Specifies the file, library, and volume names of the file to be opened on the indicated file-number. These will be requested in a GETPARM, whether or not they are named explicitly, unless NOGETPARM or NODISPLAY has been specified.

FILESEQ = exp 1 -- Specifies the file sequence number. This is to be used only for TAPE files to position the Read/Write head at the start of the correct file number for TAPE files.

8.3.3 File I/O Modes

INPUT - Files opened for INPUT may be accessed only through the READ statement and, for consecutive files, the SKIP statement. The READ statement reads consecutive files from tape and consecutive or indexed files from the disk.

I/O - Files opened for I/O may be accessed through the READ statement. If the READ statement specifies the HOLD option, the record read may be subsequently modified using the REWRITE statement (or, for indexed files, either the WRITE, REWRITE, or DELETE statements). As with INPUT, the SKIP statement is available for consecutive files.

OUTPUT/EXTEND - Files opened for OUTPUT or EXTEND can be accessed using the WRITE statement only. EXTEND mode is supported only for consecutive disk files.

SHARED - SHARED mode disk I/O is supported only for indexed files and special log files. The file may be accessed using the READ, WRITE, REWRITE, and DELETE statements. Moreover, when a program opens a file SHARED, the HOLD option is available in the READ statement. This prevents other users from attempting to modify or delete the held record until the user has modified or deleted it, has begun processing another record, or has closed the file. When that second I/O operation is completed, the HOLD is released, and other users can again access that record. If the first user modified or deleted the record, that action will take effect before other users may access the record. A program may put a HOLD on only one record at a time.

8.3.4 File I/O Buffering and the Record Area

Associated with each open file is a data buffer, maintained by DMS, which serves as an intermediate storage location for data transferred between BASIC variables and the disk or tape. The size of the buffer is normally one block, which is equal to 2048 (2K) bytes. The programmer can specify a larger buffer size with the BLOCKS clause of the OPEN statement.

In addition to the DMS buffer, there is another intermediate storage area associated with each User File Block, called the record area. The size of the record area is equal to the RECSIZE specified in the SELECT statement for that file-number.

All data transferred between programs and files must pass through both the DMS buffer and the record area for that file. Transfers between program data (receivers and expressions) and the record area, and between the record area and the DMS buffer are always done one record at a time. Transfers between the DMS buffer and the file are always performed n blocks at a time, where n equals the size of the DMS buffer specified in the BLOCKS clause of the OPEN statement (if BLOCKS is omitted, n=1).

```
program data <------> record area <------> DMS buffer <------> file
          (1 record      )    (1 record      )    (n blocks      )
          (at a time;    )    (at a time;    )    (at a time;    )
          (controlled by)     (controlled by)     (controlled by)
          (BASIC program)     (BASIC program)     (DMS           )
```

The READ and WRITE statements, depending on which forms are used, cause data transfer either between the buffer and the record area, or between the buffer and program data, through the record area. The GET and PUT statements are used to control transfer of data between program data and the record area alone. These four statements are all discussed more fully in Section 8.4, The File I/O Statements.

To illustrate the relation between file-to-buffer data transfer and buffer-to-program data transfer, consider a program which processes data from a consecutive file with fixed-length, 80-byte records. If a one-block buffer is used, READing the first record from the file loads the first block of the file into the data buffer. This 2K block contains 25 80-byte records (plus 48 unused bytes; records never span a block). Any subsequent READ or WRITE statement which accesses any of the first 25 records of the file actually causes data transfer only between program data and the buffer area, through the record area. Since all of the first 25 records are already in the DMS buffer (which is in main memory), there is no need for a time-consuming disk or tape I/O operation for every record read or written. The first time a READ or WRITE occurs involving a record outside of the first block, DMS checks to see if the contents of the buffer were modified (by a WRITE or REWRITE). If so, the contents of the buffer are rewritten to the disk or tape, replacing the original block on the storage device. The block containing the next desired record is then read into the buffer from the disk or tape. If the contents of the buffer were not modified, the next desired block is simply read into the buffer, overwriting its previous contents.

If the programmer sets BLOCKS=2 (or more) in an OPEN statement, then data will be transferred 2 (or more) blocks at a time, instead of one at a time. This decreases the frequency of calls to DMS to perform time-consuming data transfers between main memory and a peripheral storage device, but increases the amount of storage space used.

The optimal choice of buffer size in any particular case will depend on several factors. The frequency of DMS-processed disk or tape I/O operations will depend on record size and on the distribution of records to be accessed through the file, as well as on buffer size. There is also a trade-off between frequency of I/O operations (decreases with increasing buffer size) and program memory requirements (increase with increasing buffer size).

The only time a BASIC programmer must consider the DMS buffer is in using the BLOCKS clause of the OPEN statement (and this is optional). Otherwise, the operation and existence of the DMS buffer is completely transparent to the BASIC user. Therefore, in subsequent discussions of file I/O, we will generally refer to data transfers between files and record areas, ignoring the intervening DMS buffer.

8.4 <u>THE FILE I/O STATEMENTS</u>

Transfer of data between BASIC programs and files is performed by five statements: READ, GET, WRITE, PUT, and REWRITE. Records in consecutive files can be read selectively by position in the file by using the SKIP statement before a READ. The DELETE statement can be used to remove selected records from an indexed file. READ, WRITE, REWRITE, SKIP, and DELETE operations can be performed on a file only while it is open (i.e., after an OPEN statement has been executed for that file, and before a CLOSE). This section describes the way in which data is transferred between the file and the record area, and between the record area and program data. For the general forms and full discussions of the various optional clauses and modes of use of these statements, see the appropriate entries in Part 2.

8.4.1 <u>The READ Statement</u>

The READ statement causes one record to be read from the specified file into the record area for that file. READ can be used with or without a list of receivers. If a list of receivers is included in the READ statement, values are extracted one by one from the record area and assigned to the receivers, left to right. If USING is specified, the values are assigned according to the formats specified in the referenced FMT or Image (%) statement (see Section 7.4, The USING Clause and Format Control Statements, and the FMT and Image (%) entries in Part 2). Otherwise, values are assumed to be in internal format (see below). If no list of receivers is present, one record is simply read from the file to the record area, and no assignments are performed. Once in the record area, a record is available to the GET, WRITE, and REWRITE statements.

If the file being read is consecutive, the RECORDS=<u>n</u> clause can be used to specify that the <u>n</u>-th record of the file is to be read. If a READ statement on a consecutive file does <u>not</u> have the RECORDS=<u>n</u> clause, the next sequential record is read. For indexed files, the KEY clause can be used to read a record with primary or alternate key equal to a particular value. See Part 2 of this manual for further details on the READ statement.

8.4.2 <u>The GET Statement</u>

The GET statement causes values to be extracted from the record area and assigned to one or more receivers. Values are extracted from the record area and assigned according to the format in an FMT or Image (%) statement (see Section 7.4, The USING Clause and Format Control Statements, and the FMT and Image (%) entries in Part 2) referenced with the USING clause, if one is present. If USING is not specified, values are assigned according to the conventions of BASIC's internal formatting. GET is generally used to assign values to receivers after a record has been read from a file by a READ statement without a receiver list. However, if a PUT, WRITE, or REWRITE statement was executed more recently than the last READ, the record area will contain whatever record was left there by the most recent of these statements. See Part 2 for further details.

8.4.3 The WRITE Statement

The WRITE statement causes one data record to be written from the record area to the disk file. WRITE can be used with or without a list of expressions. If a list of expressions is included in the WRITE statement, their values are first packed into the record area. If USING is specified, the values are packed into the record area according to the format in the referenced FMT or Image (%) statement (see Section 7.4, The USING Clause and Format Control Statements, and the FMT and Image (%) entries in Part 2). If USING is not specified, the values are packed into the record area according to the conventions of BASIC's internal format. The contents of the record area are then written to the specified file. If the WRITE statement contains no list of arguments, then the current contents of the record area are written to the file. Generally this form of the WRITE statement would be used after data had been written into the record area by a PUT statement. If, however, a READ, WRITE, or REWRITE statement was executed more recently than the last PUT, the record area will contain whatever was left there by the most recent of these statements.

Records written to consecutive files (in OUTPUT, SHARED, or EXTEND mode, as specified in the OPEN statement) are added to the end of the file. Records written to indexed files (in IO mode) are inserted into the file at the appropriate point as determined by their primary key values. See Part 2 for further details.

8.4.4 The PUT Statement

The PUT statement causes the values of one or more expressions to be packed into the record area of the specified file. If USING is specified, the values are packed into the record area according to the format in the referenced FMT or Image (%) statement (see Section 7.4, The USING Clause and Format Control Statements, and the FMT and Image (%) entries in Part 2). If USING is not specified, the values are packed into the record area according to the conventions of BASIC's internal format. PUT is generally used prior to a WRITE statement with no argument list.

8.4.5 The REWRITE Statement

REWRITE is like WRITE except that the record which is written to the file overwrites the last record read with a HOLD option, instead of being written to the end of a file. For a description of the HOLD option, see the entry under READ in Part 2. REWRITE cannot be performed on consecutive files with variable-length records.

8.4.6 Summary of Data Flow Controlled by File I/O Statements

```
                program variables          record      ( DMS  )      disk or tape
                 or expressions             area       (buffer)          file

READ            X (<---- optional -----) X <-------- (X) <------------ X

GET             X  <------------------  X

(RE)WRITE       X (----- optional ---->) X --------> (X) ------------> X

PUT             X -------------------->  X
```

8.4.7 Data Representation in File I/O

In using file I/O statements, it is important to keep in mind that BASIC represents numeric data in an internal format which bears no simple relation to the sequences of ASCII character codes used to represent those same data in workstation or printer I/O. Unless file I/O is explicitly formatted with the USING clause and Image (%) or FMT statements, all file I/O is performed in this internal format. While this is suitable for data files which are to be read only by other programs, it should not be used for files to be directly examined by users, such as report or other text files. Any attempt to display or print numeric data from a file in internal format will simply produce meaningless strings of characters on the workstation or printer.

Data values packed into records in internal format take up the following amounts of space:

Floating-point -- 8 bytes
Integer -- 4 bytes
Alphanumeric -- defined length

If any format conversions are done (i.e., if USING is specified in any file I/O statement), they are performed as data are transferred between the record area and variables or expressions in the program. Data should always be read from a file in the same format in which they were written in order to be properly interpreted by a BASIC program.

For a discussion of the FMT and Image (%) statements, see Section 7.4, The USING Clause and Format Control Statements, and the FMT and Image (%) statement entries in Part 2.

8.5 INTRINSIC FILE I/O FUNCTIONS

Four functions may be used in expressions to retrieve information concerning file I/O operations: FS, KEY, MASK, and SIZE.

8.5.1 FS (file-expression)

The FS function returns the file status for the most recent I/O operation on the specified file, as an alpha value two characters long. FS can assume any of the following values:

CONSEC, TAPE, and PRINTER file I/O

'00'	Successful I/O operation
'10'	End-of-file encountered
'23'	Invalid record number
'30'	Hardware error
'34'	No more room in the file
'95'	Invalid function or function sequence
'97'	Invalid record length

INDEXED file I/O

'00'	Successful I/O operation
'10'	End-of-file encountered
'21'	Key out of sequence (WRITE statement in OUTPUT mode only)
'22'	Duplicate key
'23'	No record found matching specified key
'24'	Supplied key exceeds any key in the file (INPUT, I/O, or SHARED mode)
'34'	No more room in the file (OUTPUT or EXTEND mode)
'30'	Hardware error
'95'	Invalid function or function sequence
'97'	Invalid record length

8.5.2 KEY (File-expression [, exp])

The KEY function returns the value of a key field from the specified file's record area. The file-expression given as the argument of the KEY function must refer to an INDEXED file. The optional second expression is a key number specifying which key field is desired. If it is omitted or set equal to zero, the primary key is returned. Otherwise, the specified alternate key (as defined in the SELECT statement) is returned. The KEY function is typically read immediately following a READ statement (i.e., without any intervening WRITE statement).

The KEY function returns an alpha value, whose length is equal to the value of the KEYLEN parameter in the SELECT statement for that file.

The KEY function may be used as a receiver in order to write into the "key" field in the data buffer.

```
3900 LET KEY(#3)="NYC"
```

KEY is typically used in this way immediately preceding a READ, WRITE or REWRITE statement. The use of arguments in the WRITE, REWRITE, and PUT statements causes data to be loaded into the data buffer. Depending on the size of the argument list and the position of the key field, loading the data buffer through arguments to WRITE, REWRITE, or PUT may overwrite the key written into the data buffer by the "LET KEY(#n)=" construction.

8.5.3 MASK (File-expression)

The MASK function returns the alternate key access mask for the last record read from the alternate indexed file specified. The result is a 2-byte alpha HEX value whose component bits (left to right) correspond to the record's available alternate keys (1-16). Bits which are "on" (binary 1) specify that the record may be READ by those alternate key paths. The bit values may be determined by printing, in hexadecimal, the result of the MASK function.

For example, if the program fragment

```
300 READ #1, PLEXIPPUS$
400 DIM A$2
500 A$ = MASK(#1)
600 PRINT HEXOF(A$)
```

were to read a record accessible by alternate keys 1, 3, 5, and 7, then line 600 would print (or display)

```
AA00
```

which represents the binary string 1010101000000000, indicating that the first, third, fifth, and seventh alternate keys are used in this record.

The MASK function may also be used as a receiver to set the alternate key access mask for a record which is to be written (or re-written). For example,

```
2300 MASK(#DESTINATION) = 6400
```

would cause the next record written to the specified indexed file to be accessible by alternate keys 2, 4, and 6 (6400 hex = 0101010000000000 binary). All records written to this file will have the same alternate key access mask until another mask value is assigned in this way.

8.5.4 SIZE (File-expression)

The SIZE function returns as an integer the size in characters of the record most recently read from the specified file.

8.6 ERROR RECOVERY

The situations under which an Input/Output instruction cannot be successfully completed fall into four categories:

1. Errors handled by the VS Data Management System. There is an error or omission in the specification of a file, library, or volume name: the file was not found, the volume is not mounted, a name was omitted, etc.

2. EOD errors. There is no more data in the file to read, or an attempt was made to write a record with a duplicate key to an indexed file. These are errors corresponding to FS codes '10' through '24' (see Section 8.5, Intrinsic File I/O Functions).

3. DATA errors. The data conversion routines failed because a record format was illegal; for instance, the program tried to read "ABC" into a numeric variable using a format such as ###. These are errors which occur within the BASIC program; since they do not occur at the stage where data is actually transferred to or from a file, they do not change the File Status (FS) code for that file.

4. IOERR errors. Other input/output errors, such as physical errors operating the device, record-length errors, and file boundary errors. These are errors corresponding to FS codes "30" through "99."

The Data Management System attempts to resolve some I/O errors of the first category by means of a dialogue with the workstation operator at the time of the error. BASIC allows the user to specify program branches to be taken if a type EOD, DATA, or IOERR error occurs. Either a GOTO or a GOSUB exit may be used. If GOSUB is used, a RETURN statement at the end of the subroutine will return program execution to the statement following the file I/O statement which had the error.

To specify error branching, the programmer specifies (1) the type of error situation to be covered (EOD, DATA, or IOERR), (2) the type of transfer of control to be performed (i.e., returning (GOSUB) or nonreturning (GOTO)), and (3) the BASIC line number or statement label to which control is to be passed. For instance, to force a returning branch to the statement labeled TURTLE if a data conversion error occurs, the programmer writes:

```
DATA GOSUB TURTLE
```

in the READ or WRITE statement.

Error branches for type IOERR errors are specified in the SELECT statement. Any IOERR errors which occur on a given file number must transfer control to a single routine. Error branches for type DATA errors are specified in the READ or WRITE statement. Different statements may transfer to different service routines in the event of a data conversion error. Error branches for EOD error conditions may be specified in a SELECT statement to apply to all reads and writes under that file number, or they may be specified in an individual READ or WRITE statement to apply to errors occurring as a result of that individual statement. If a READ or WRITE statement has an EOD exit, that exit overrides any transfer of control which may have been specified in the SELECT statement.

In addition, the REWRITE, PUT, and GET statements can specify an error branch for DATA type errors. The SKIP statement can specify an error branch for EOD type errors (which would occur if an attempt were made to SKIP past the limits of the file).

If an EOD, DATA, or IOERR type error occurs and the program has not specified an error branch, execution of the program is aborted.

The service routines for EOD and IOERR type errors may examine the expression FS(#n), which returns the file status for the file currently open on UFB #n, to determine the exact cause of the error.

8.7 EXAMPLES OF FILE I/O

Consider a program which takes as input a consecutive file containing a list of names, addresses, and phone numbers. Each record of the file contains the following information in the following positions:

Name:	bytes 1 - 20
Street:	bytes 21 - 40
City:	bytes 41 - 50
State:	bytes 51 - 52
Zip Code:	bytes 53 - 57
Area Code:	bytes 58 - 60
Phone:	bytes 61 - 67

This program produces as output an indexed file containing those records which have their state fields (bytes 51 - 52) equal to "MA".

```
 100 SELECT #1, "INPUT", CONSEC, RECSIZE=67, EOD GOTO NO_MORE
 200 SELECT #2, "OUTPUT", INDEXED, RECSIZE=67, KEYPOS=1, KEYLEN=20
 300 DIM REC$ 67
 400
 500 /* OPEN AND CLOSE INDEXED FILE TO CREATE IT SO THAT ENTRIES CAN
 600    BE WRITTEN IN ANY ORDER */
 700 OPEN #2, OUTPUT, SPACE=100, FILE="BOSTON", LIBRARY="ADDRESS",  !
 800   VOLUME="DATA"
 900 CLOSE #2
1000
1100 /* OPEN BOTH FILES */
1200 OPEN #1, INPUT, FILE="USA", LIBRARY="ADDRESS", VOLUME="DATA"
1300 OPEN #2, IO, FILE="MASS", LIBRARY="ADDRESS", VOLUME="DATA"
1400
1500 GET_RECORD:
1600 READ #1, REC$      /* READ A RECORD FROM THE CONSEC FILE   */
1700 IF STR(REC$,51,2)<>"MA" THEN GET_RECORD  /* EXAMINE STATE  */
1800 WRITE #2, REC$     /* WRITE TO INDEXED FILE IF STATE="MA"  */
1900 GOTO GET_RECORD    /* GET ANOTHER RECORD */
2000
2100 NO_MORE:           /* EXIT ROUTINE FOR EOD ON FILE #1 */
2200 CLOSE #1
2300 CLOSE #2
2400
2500 END
```

The two SELECT statements describe the two files to be used: file #1 is consecutive, file #2 is indexed; both have fixed-length, 67-byte records. When and if an end-of-data (EOD) condition occurs on file #1, control will pass to the statement labeled NO_MORE. The primary key field for the indexed file (#2) begins at the first byte of each record, and is 20 bytes long.

The first time an indexed file is opened for output (i.e., when it is created), any records written to it must be written in primary key sequence. If the records to be written to an indexed file are not in order the first time the file is to have data written to it, the file must be opened and closed in OUTPUT mode without writing any records to it, thus creating a file with zero records in it (lines 700 - 900). It can then be re-opened in IO mode (line 1300), which allows records to be written in any order.

NOTE:

In general, it is preferable to write records to indexed files in key sequence, if possible. Writing records to a new indexed file out of sequence is much less efficient in terms of both processor time and disk space, and is recommended only when it is not practical to write records in sequence.

Once both files are opened, a record (REC$) is obtained from the consecutive file (line 1600) and the two characters of its state field are tested to see if they are equal to "MA" (line 1700). If so, the record is written to the indexed file (line 1800); if not, the next record is read from the consecutive file (label GET_RECORD; line 1500). This cycle continues until all of the records in the consecutive file have been read. The first READ operation after the last record has been read causes an EOD error condition to occur, and control passes to the statement labeled NO_MORE (line 2100), as specified in the EOD clause of the SELECT statement. Both files are then closed (lines 2200 - 2300), and the program ends.

Note that the first twenty bytes (the name field) of each record are designated as the primary key field for the indexed file in the SELECT statement for that file. Any subsequent read from the file MASS by primary key would then obtain the address records in alphabetical order of addressees' names. However, they may have been written in any order; the order in which they were written was determined simply by the order of their appearance in the consecutive file, which was arbitrary. (Recall that the indexed file was opened and then closed without writing any records, and then re-opened again, specifically to enable the program to write the records in any order.)

Suppose that one planned to make sorted lists at some later time based upon the telephone area codes and zip codes of the addressees. Such sorting could be simplified by establishing alternate keys in the indexed file corresponding to the area code and zip code fields of the records. This could be done by changing the SELECT statement for the indexed file to:

```
200 SELECT #2, "OUTPUT", INDEXED, RECSIZE=67, KEYPOS=1, KEYLEN=20,     !
220   KEYPOS=1, KEYLEN=20,                     /* PRIMARY KEY = NAME */  !
240   ALT KEY 1, KEYPOS=58, KEYLEN=3, DUP,   /* KEY 1 = AREA CODE  */    !
260       KEY 2, KEYPOS=53, KEYLEN=5, DUP    /* KEY 2 = ZIP CODE   */    !
```

The file will now have two alternate indices: alternate index 1, which indexes records by the three-byte field starting at byte 58 of the record (the area code field, according to the convention above), and alternate index 2, which indexes records by the five-byte field starting at byte 53 (the zip code field). Both indices allow duplicate keys (DUP) since there may be more than one entry with the same area or zip code.

In order to insure that the records written to the file are retrievable later by these alternate keys, the alternate key access mask must be set appropriately before any records are written. The usable keys are numbered 1 and 2, so the binary value of the alternate key access mask should be set to 1100000000000000, which is C000 in hexadecimal (see discussion of MASK function in Section 8.5, Intrinsic File I/O Functions). So, line 1800 can be changed to read:

```
1800 WRITE #2, MASK=HEX(C000), REC$
```

To produce a consecutive file containing all the records from the MASS file which have have their area code fields equal to "617", one could use the following program (assuming the file MASS were written with alternate keys as just described):

```
 100 SELECT #1, "INPUT", INDEXED, RECSIZE=67,                         !
 200   KEYPOS=1, KEYLEN=20,                     /* PRIMARY KEY=NAME */!
 300   ALT KEY 1, KEYPOS=58, KEYLEN=3, DUP,     /* KEY 1=ZIP CODE   */!
 400       KEY 2, KEYPOS=53, KEYLEN=5, DUP      /* KEY 2=AREA CODE  */
 500 SELECT #2, "OUTPUT", CONSEC, RECSIZE=67
 600
 700 DIM REC$ 67
 800
 900 OPEN #1, INPUT, FILE="MASS", LIBRARY="ADDRESS", VOLUME="DATA"
1000 OPEN #2, OUTPUT, SPACE=200, FILE="AREA617", LIBRARY="ADDRESS",!
1100   VOLUME="DATA"
1200
1300 FIRST_IN:        /* GET FIRST RECORD W/ALT KEY 1 = "617" */
1400 READ #1, KEY 1 = "617", REC$, EOD GOTO THE_END
1500 GOTO NEXT_OUT
1600
1700 NEXT_IN:         /* GET NEXT RECORD W/ALT KEY 1 = "617" */
1800 READ #1, REC$, EOD GOTO THE_END
1900 IF KEY(#1,1) <> "617" THEN THE_END
2000
2100 NEXT_OUT:        /* WRITE THE RECORD OUT TO THE CONSEC FILE */
2200 WRITE #2, REC$
2300 GOTO NEXT_IN
2400
2500 THE_END:
2600 CLOSE #1
2700 CLOSE #2
2800 END
```

In this case the indexed file MASS is associated with the file number (UFB) #1. Note that the attributes specified in the SELECT #1 statement (line 100) are exactly the same as those specified in the SELECT statement in the program which created the file MASS.

After the alternate indexed file MASS and the consecutive file AREA617 are opened (lines 800 - 1000), records with alternate key 1 (area code field) are read from MASS one at a time (lines 1300, 1700) and written to AREA617 (line 2000). Note that the program has two separate routines (labeled FIRST IN and NEXT_IN) for reading the first and subsequent records. This is because any statement of the form READ #n, KEY m = alpha-exp reads only the first occurence in the file of a record with alternate key m equal to alpha-exp. Any subsequent READ #n statement which does not specify an alternate key number will read the next occurence of a record with the most recently specified alternate key. The most recently specified alternate key path (m) is the current "reference key" for a particular indexed file. To change the reference key for a file it is necessary to execute a READ with an explicitly specified key. The primary key is considered to be key 0 (zero).

After each record is read, its first alternate key field is examined (line 1900). The first time a record is read with alternate key 1 not equal to "617", the program branches to the statement labeled "THE_END". Both files are then closed (lines 2600 - 2700) and the program ends.

CHAPTER 9
DATA CONVERSION AND MATRIX STATEMENTS

9.1 DATA CONVERSION STATEMENTS

VS BASIC provides an extensive set of instructions designed specifically to simplify the task of converting data from one format to another, either for the purpose of interpreting information in a foreign format, or for packing data into a more efficient format for storage or transmission. The statements included in this special data conversion instruction set are summarized below:

CONVERT -- Converts a numeric value to an alphanumeric character string and vice versa.

HEXPACK, HEXUNPACK -- HEXPACK converts a character string representing hexadecimal digits into the binary equivalent of the digits. HEXUNPACK does the reverse.

ROTATE[C] -- Rotates the bits of a single character or a string of characters.

TRAN -- Utilizes a table-lookup technique to provide high-speed character conversion.

These statements are discussed at length under their individual entries in Part 2.

In addition to the above statements, other VS BASIC instructions which may be useful in data conversion operations include the Boolean operations AND, OR, XOR, and BOOLh (discussed in Section 5.7, Logical Expressions), the alphanumeric functions BIN and VAL (discussed in Section 5.5, Alphanumeric Functions, Section 5.6, Numeric Functions With Alpha Arguments, and under their entries in Part 2), and the binary arithmetic operations ADD and ADDC (discussed in Section 5.7, Logical Expressions, and under their entries in Part 2).

9.2 MATRIX STATEMENTS

VS BASIC offers a set of matrix statements which perform operations upon entire arrays. The matrix statements provide fifteen built-in matrix operations, summarized by function below. Detailed discussions of each can be found in Part 2.

9.2.1 Matrix I/O Statements

MAT INPUT -- allows run-time input of numeric or alphanumeric array values.

MAT PRINT -- Displays or prints one or more arrays. Matrices are printed row-by-row.

Both MAT INPUT and MAT PRINT allow explicit redimensioning of arrays (see Subsection 9.2.4, Array Dimensioning).

9.2.2 Matrix Assignment Statements

MAT CON -- Sets every element of a numeric array to 1.

MAT= -- Replaces each element of a numeric or alphanumeric array with the corresponding element of a second array. The first array is redimensioned to conform to the second.

MAT IDN -- Causes a (square) matrix to assume the form of the identity matrix.

MAT READ -- Assigns values contained in DATA statements to array variables without referencing each member of the array individually.

MAT TRN -- Causes a numeric or alphanumeric array to be replaced by the transpose of a second array. The first array is redimensioned to correspond to the transpose of the second.

MAT ZER -- Sets every element of an array to zero.

All of the matrix assignment statements listed above allow explicit redimensioning of arrays. See Subsection 9.2.4, Array Dimensioning.

9.2.3 Matrix Arithmetic and Sorting Statements

MAT + -- Adds two numeric arrays of the same dimension.

MAT - -- Subtracts numeric arrays of the same dimension.

MAT ()* -- Multiplies each element of a numeric array by an expression.

MAT * -- Stores product of two numeric arrays in a third array.

MAT INV -- Replaces one numeric matrix by the inverse of another.

MAT ASORT, MAT DSORT -- Sorts one alphanumeric or numeric array in ascending or descending order into a second array.

Operations are performed on numeric arrays according to the rules of linear algebra and can be used for the solution of systems of non-singular homogenous linear equations. Inversion of matrices can be done in significantly shorter time than is possible with ordinary BASIC statements. MAT operations on alphanumeric arrays can be used for simple and rapid I/O (input/output) and printing of alphanumeric material.

Note that the arithmetic and sorting statements described above do not allow explicit redimensioning of arrays.

9.2.4 Array Dimensioning

Both numeric and alphanumeric arrays may be manipulated with MAT statements. If not dimensioned in a DIM or a COM statement, arrays are given default dimensions of 10 by 10, with a default alphanumeric element length of 16 bytes. Each dimension may range from 1 to 32,767 with an alpha element length of 1 to 256 bytes.

The dimensions of an array may be changed explicitly using the MAT REDIM statement. This may also be done by giving the new dimensions, enclosed in parentheses, following the array name in any of the following MAT statements:

```
MAT CON
MAT IDN
MAT INPUT
MAT READ
MAT ZER
```

Arrays may also be redimensioned implicitly, as shown in the following example.

```
100 DIM A(10,10),B(2,2),C(2,2)
200...
400 MAT A=B+C
```

The array A is redimensioned at statement 400 from a 10 x 10 array to a 2 x 2 array.

For alphanumeric arrays, the maximum length of each element may be changed by specifying the new length after the dimension specification. For example:

```
MAT REDIM A$(2,3)10
```

redimensions the array A$ to be two rows by three columns with the maximum length of each element in the array equal to 10.

NOTE:

With either explicit or implicit redimensioning, the newly dimensioned array must not require more space than was required for its original dimensions. For numeric arrays, this implies the same (or fewer) elements. For alphanumeric arrays, there must be the same number (or fewer) characters.

9.2.5 Matrix Statement Rules

The following rules must be observed in the use of matrix statements:

1. Each matrix statement must begin with the word MAT.

2. Multiple matrix operations are not permitted in a single MAT statement. For instance, MAT A = B+C-D is invalid. The same result can be achieved by using two MAT statements: MAT A = B+C and MAT A = A-D.

3. Arrays which contain the result of certain MAT statements are automatically redimensioned; other arrays can be redimensioned explicitly in the MAT REDIM statement. A redimensioned numeric array cannot contain more elements than given in its original definition; a redimensioned alphanumeric array also cannot contain more characters than given in its original definition.

4. A vector (a singly-subscripted array) cannot be redimensioned as a matrix (a doubly-subscripted array), nor can a matrix be redimensioned as a vector.

5. The same array variable cannot appear on both sides of the equation in matrix multiplication, matrix transposition, or matrix sorting. MAT C=A*B and MAT A=TRN(C) are valid MAT statements; MAT C=C*B and MAT B=TRN(B) are not.

PART 2
VS BASIC STATEMENTS AND FUNCTIONS

The following rules are used in this manual in the syntax specifications to describe BASIC program statements and system commands:

1. Uppercase letters (A through Z), digits (0 through 9), and special characters (*, /, +, etc.) must be written exactly as shown in the general form.

2. Lowercase words represent items which are supplied by the user.

3. Items in square brackets ([]) indicate that the enclosed information is optional. For example, the general form: RESTORE [expression] indicates that the RESTORE statement can be optionally followed by an expression.

4. Braces ({}) enclosing vertically stacked items indicate alternatives; one of the items is required. For example,

   $$\text{operand} = \begin{Bmatrix}\text{literal}\\ \text{alpha variable}\\ \text{expression}\end{Bmatrix}$$

   indicates that the operand can be either a literal, an alpha variable, or an expression.

5. Ellipsis (...) indicates that the preceding item can be repeated as necessary. For example,

   ```
   INPUT [literal,] receiver [,receiver]...
   ```

   indicates that additional receivers can be added to the INPUT statement as needed.

6. The order of parameters shown in the general form must be followed.

ABS Function

General Form:

ABS(numeric expression)

ABS returns the absolute value of the numeric expression specified as its argument. The value returned by the ABS function is of the same type (integer or floating-point) as the argument.

Program Example:

```
10  A = 47
20  B = -A
30  Print A, B, ABS(B)
```

Result:

```
47      -47      47
```

ACCEPT Statement

```
General Form:

ACCEPT   list      [,list]...

      [,KEYS(alpha-arg1)] [,KEY(numeric variable)]

      [,ON alpha-arg2 {GOTO } {line number    } [,{line number    }]...]
                      {GOSUB} {statement label} [ {statement label}]

      [ {,ALT                                      } ]
      [ {                                          } ]
      [ {,NOALT{ GOTO} { line number    }          } ]
      [ {      {GOSUB} {statement label}           } ]

where:
    list=  AT (exp2, exp3)
           literal
           [FAC(alpha-arg3),]{num variable   [, PIC(image)][,num-spec]  }
                             {alpha variable [, CH(int)][,alpha-spec]   }


    num-spec =  RANGE    {(POS)        }
                         {(NEG)        }
                         {(exp4, exp5) }

    alpha-spec ={RANGE    (alpha-arg4, alpha-arg6)}

    image    = a valid numeric image, as in FMT.

    int      = an int specifying the length of the (alpha) field.

    alpha-arg= literal, alpha variable, BIN function, STR function.
```

The ACCEPT statement is discussed in detail in Section 7.5, the ACCEPT statement. The following section is a summary of the features and operation of ACCEPT.

The ACCEPT statement allows workstation input of numeric and alphanumeric data in a field-oriented manner, using the supplied formatting information. Both single receivers and arrays may be input.

ACCEPT uses the entire screen, clearing all unused areas.

Field Descriptions

1. Numeric fields may be formatted according to the PIC() specification. It is interpreted as in the FMT statement (see FMT statement). If PIC() is omitted, the numeric fields are 18 characters. All blanks appear on the screen as pseudoblanks.

2. Alphanumeric field width is specified by CH(int), where int = field width. If CH is omitted, the field size defaults to the defined length of the alpha value. All blanks appear as pseudoblanks on the screen.

Field Attribute Characters (FAC's)

1. If omitted, the following defaults are assumed:

   Alphanumeric -- bright, modifiable, uppercase, tabbable (HEX(81)).

   Floating-point -- bright, modifiable, uppercase, tabbable (HEX(81)).

   Integer -- bright, modifiable, numeric only, tabbable (HEX(82)).

2. The first character of the alpha-expression specified in the FAC clause (alpha-arg3) is used as the FAC character.

Field Placement Order

1. For single receivers, the fields are placed one at a time in order of appearance in the statement, or in the order implied by any AT clauses which are used.

2. For arrays, the fields are arranged element-by-element, in the usual row-by-row order (like MAT PRINT).

Field Positioning

A field can be explicitly placed at a specified row and column on the screen, using the AT clause of the ACCEPT statement; if no AT clause is given, the field will be placed according to the defaults used by ACCEPT, which are as follows:

1. If the field can fit on the same line as the preceding field, the field will follow directly after the preceding field with space for one FAC left between the fields. If the field in question is the first field on the screen (i.e., there is no preceding field), then the field is placed by default at row 1 column 2, to leave room for a preceding FAC.

2. Any modifiable field which is too long to fit in the space remaining on the line containing the preceding field will be placed at the beginning of the second column of the next line on the screen. No modifiable field can be too long to fit on a single line (79 bytes maximum length).

3. Any non-modifiable field too long to fit in the space remaining on the line which has the preceding field and which is no longer than 79 bytes, is placed at the beginning of the second position on the following line. If it is longer than 79 bytes, it is placed immediately following the preceding field, and will be continued onto as many lines as necessary.

4. If a non-modifiable field is too long to fit completely on the line on which it starts, it will be continued for as many lines as necessary. Each new line will begin with a FAC with the same attributes as the FAC which comes at the beginning of the field, except that the continued sections of the field will not be tabbable.

   These rules are summed up in Table P2-1.

The following conditions are considered errors, whether they occur because the field was placed using an AT clause, or because the field was placed by the ACCEPT defaults:

1. Any modifiable field longer than 79 bytes (too long to fit on a single line).

2. Any explicitly positioned modifiable field extending beyond the end of the line on which it is placed.

3. Any field explicitly placed so that it starts beyond the boundaries of the screen.

4. Any field extending beyond the end of the last line on the screen.

For arrays, the cursor is automatically moved to column 2 of the next line on the screen after each row.

Table P2-1.

ACCEPT Field Placement Defaults

LINE LENGTH	MODIFIABLE FIELD	NON-MODIFIABLE FIELD
Less than 79 characters		
will fit on line	Immediately follows previous field	Immediately follows previous field
won't fit on line	Begins on next line	Begins on next line
More than 79 characters	Not allowed	Immediately follows previous field

Validation

Data entered by the user in response to an ACCEPT screen may be validated by either character type or value. Character type validation is controlled by the FAC clause. The FAC which precedes a field determines which types of characters (i.e., numeric only, all characters, uppercase only) may be typed in that field. Attempting to type in any character prohibited by that field's FAC causes the workstation alarm to sound, and the character is ignored.

Both numeric and alphanumeric fields may be validated by the BASIC program, according to a specified range of values, before being accepted. If validation fails, the first incorrect field is set to "blinking" and the user is reprompted for the values. Validation is done via a range specification as follows:

1. Numeric

   RANGE: POS = positive values (including zero).
   NEG = negative values only.
   exp4, exp5 = lower and upper limits, respectively, for the input value(s) (inclusive). If a negative value is specified for a limit the expression must be placed in parenthesis.

2. Alphanumeric

   RANGE: alpha expl, alpha exp2 = lower and upper limits. The ASCII collating sequence is used.

PF Key Control

The action taken by the program in response to ENTER and PF keys can be controlled by any combination of three key control clauses. (PF keys in ACCEPT statements do not call DEFFN' subroutines or strings.)

If all three clauses are omitted, only the ENTER key can be used to respond to the ACCEPT. If any clause is present, ENTER and all PF keys are allowed by default, subject only to the restrictions of the KEYS clause if present.

> KEYS -- This clause specifies the keys which are valid for this ACCEPT; any others will sound the workstation alarm if pressed. The alpha- expression (actual length) is used as a list of 1-byte binary values corresponding to the allowed PF key (ENTER = 00). Invalid values are ignored. (PF32 = HEX(20) may be considered to be a trailing blank if the user is not careful.) The key order is irrelevant.
>
> KEY -- This causes the number of the key (ENTER = 0) pressed by the user to be placed in the numeric variable. This is done prior to any field validation or exit branching. The KEYS clause takes precedence over the KEY clause.
>
> ON Key Value -- This clause allows the user to exit without changing any data values if certain PF keys are specified.
>
> As in the KEYS clause, the alpha-expression (actual length) is treated as a PF key list. Each entry in the list corresponds to a line number or statement label to which the program branches if that PF key is pressed.
>
> The last line number should not be followed by a comma, nor should unused line numbers or statement labels be specified.

Response to Modification of Data

1. Ordinarily, all modifiable fields are read/validated/transferred to their receivers, whether or not the fields were actually changed by the user.

   This can be made more efficient via the ALT specification or NOALT clause. The presence of either ALT or the NOALT exit in the ACCEPT statement will cause only those fields which were altered by the user (i.e., character keystrokes detected at the workstation) to be processed. Unaltered fields are effectively ignored, and the corresponding receivers are unchanged.

2. If NOALT is specified and no fields were altered, the specified exit is taken.

3. If ALT is specified, only those fields which were altered will be processed; however, no exit may be specified.

Execution of ACCEPT

1. The screen is generated as described, with the cursor positioned at the first modifiable (or numeric-protected) field, if any. All fields contain the current values of the receivers/array elements.

2. The user may enter new values. When ENTER is keyed, or a PF key is pressed, the key is first checked for validity. If invalid, the workstation alarm sounds, and the user may continue modifying or may press another key.

3. If the key is specified in the ON clause, the specified branch is taken without any field reads or verification. (The KEY variable will contain the key number, in any case.)

4. Otherwise, all modifiable fields (or only altered fields if ALT or NOALT is specified) are read/validated. Numeric fields are validated for proper numeric format independently of RANGE validation. Although any PIC specification may be used, special characters (CR,DB, etc.) are not valid on input.

   If any field is invalid, its FAC is set to blinking and the user must correct the mistake (and can further change other fields).

Syntax Example:

```
300 ACCEPT AT (12, 15), A, PIC(###), RANGE(50,100),                 !
310   FAC(HEX(91)), B$, CH(7), RANGE("BARRELS", "KEGS"),             !
320   "OF BEER ON THE WALL.",                                        !
330   KEYS(BIN(0) & BIN(1) & BIN(16)), KEY(OPTION),                  !
340   ON (BIN(1) & BIN(16)) GOTO START, FINISH,                      !
350   NOALT GOSUB 1700
```

ADD[C] Logical Operator

```
General Form:

[LET] alpha-receiver = [logical expression] ADD[C] logical expression


   logical expression:  see Section 5.7, Logical Expressions.
```

The ADD operator is used to add a binary value to the binary value of an alpha variable. For example, in the statement

```
100 A$ = ADD B$
```

the binary value of B$ is added to the binary value of A$, and the result is stored in A$.

If an operand is specified before the ADD operator (operand-1), its value is stored in the receiver variable prior to performing the addition. For example, in the statement

```
100 A$ = C$ ADD B$
```

the value of C$ is first stored in A$; the value of B$ is then added to A$, and the result stored in A$. The contents of operand-1 and the operand which follows the ADD operator (operand-2) are not altered.

If C does not follow the ADD operator, the addition is carried out on a character-by-character basis from right to left, with no carry propagation between characters. That is, the rightmost byte of the value of the operand is added to the rightmost byte of the receiver variable; then, the next-to-last character of the operand is added to the next-to-last character of the receiver, and so forth. For example:

```
100 DIM A$2
200 A$=HEX(0123)
300 A$=ADD HEX(00FF)
400 PRINT "RESULT = ";HEXOF(A$)

Output:  RESULT = 0112
```

If the operand and receiver are not of the same defined length, the shorter one is left-padded with hex zeros. The result is right-justified in the receiver, with high-order characters truncated if the result is longer than the receiver.

If C does follow ADD, the value of the operand is treated as a single binary number and added to the binary value of the receiver variable with carry propagation between characters.

For example:

```
100 DIM A$2
200 A$=HEX(0123)
300 A$=ADDC HEX(00FF)
400 PRINT "RESULT = ";HEXOF(A$)

OUTPUT:  RESULT = 0222
```

<u>Examples of of valid syntax</u>

```
600 A$=ADD HEX(FF)

200 A$=ADDC ALL(FF)

900 STR(A$,1,2)=B$ ADDC C$
```

See Section 5.7, Logical Expressions, for more information.

ALL Function

General form:

ALL (alpha-expression)

The ALL function creates a string consisting entirely of characters equal to the first character of the alpha-expression, and has a length equal to the defined length of the receiver. It is used only in logical expressions. (For more information on the use of the ALL function, see Section 5.7, Logical Expressions.)

Syntax examples:

```
400 LET A$=ALL(B$)

800 C$=AND ALL(D$)
```

AND Logical Operator

```
General Form:

[LET] alpha-receiver = [logical exp] AND logical exp


   logical exp:  see Section 5.7, Logical Expressions.
```

The AND operator logically AND's two or more alphanumeric arguments. Redefine w/out using the term.

The operation procedes from left to right. If the operand (the logical expression) is shorter than the receiver, the remaining characters of the receiver are left unchanged. If the operand is longer than the receiver, the operation stops when the receiver is exhausted.

Examples:

100 A$ = AND B$ (logically ANDs A$ and B$ and places the result in A$.)

100 A$ = B$ AND C$ (logically ANDs B$ and C$ and places the result in A$.)

```
HEX(0F0F) AND HEX(0F0F)=HEX(0F0F)
HEX(00FF) AND HEX(0F0F)=HEX(000F)
```

See Section 5.7, Logical Expressions, for more information.

ARCCOS Function

General Form:

```
ARCCOS(numeric expression)
```

The ARCCOS function returns the arccosine of its argument; this is the inverse function of COS. The value of the numeric expression used as an argument to ARCCOS must be between 0 and 1 (inclusive); otherwise, an error message will result when the ARCCOS function is evaluated and program execution will halt. ARCCOS returns a floating-point value in radians, degrees, or grads, depending on the trig mode specified by the most recently executed SELECT statement (the default is radians if no SELECT has been executed).

ARCSIN Function

```
General Form:

ARCSIN(numeric expression)
```

The ARCSIN function returns the arcsine of its argument; this is the inverse function of SIN. The value of the numeric expression used as an argument to ARCSIN must be between 0 and 1 (inclusive); otherwise, an error message will result when the ARCSIN function is evaluated and program execution will halt. ARCSIN returns a floating-point value in radians, degrees, or grads, depending on the trig mode specified by the most recently executed SELECT statement (the default is radians when no SELECT has been executed).

ARCTAN Function

```
General Form:

ARCTAN(numeric expression)
```

The ARCTAN function returns the arctangent of its argument; this is the inverse function of TAN. ARCTAN returns a floating-point value in radians, degrees, or grads, depending on the trig mode specified by the most recently executed SELECT statement (the default is radians when no SELECT has been executed).

ATN Function

General Form:

```
ATN(numeric expression)
```

The arctangent function; synonymous with ARCTAN.

BIN Function

```
General Form:

BIN(expression [,d])

where:
    d = 1,2,3,4   (default = 1).
```

This function converts the integer value of the expression to a d-character alphanumeric string, which contains the binary equivalent of the expression. BIN is the inverse of the function VAL.

For d = 1, 2, or 3, the expression is converted to a d-byte unsigned binary number. The limits for the value of the expression are:

$$0 \le \text{val. expression} < \left\{\begin{matrix} 256 & (d=1) \\ 65536 & (d=2) \\ 16777216 & (d=3) \end{matrix}\right\}$$

For d=4, the expression is converted to a 4-byte 2's-complement signed binary number (like internal integer format). The range is -2147483648 $\le$ val of expression $\le$ 2147483647

Syntax examples:

```
300 A$=BIN(A,4)

800 B$=BIN(A,3) AND BIN(B,3)
```

Numeric examples:

```
BIN (255,1) = HEX(FF)

BIN (65535,2) = HEX (FFFF)

BIN (32767,3) = HEX (007FFF)
```

BOOLh Logical Operator

```
General Form:

[LET] alpha-receiver = [logical exp] BOOLh logical exp


   logical exp - see Section 5.7, Logical Expressions.
   h = a digit from 0 to 9, or a letter from A to F.
```

BOOL is a generalized logical operator that performs a specified operation on the value of the receiver alpha variable. The operation to be performed is specified by the hexadecimal digit following BOOL (see Table P2-2). BOOL may be used only in the alpha-expression portion of an assignment statement (i.e., on the right-hand side of the equals (=) sign). The value of the operand which follows the BOOLh operator (operand-2) and the value of the receiver variable are operated upon, and the result is stored in the receiver variable. For example, the statement

```
100 A$ = BOOL7 B$
```

logically not-AND's the value of B$ with the value of A$, and stores the result in A$.

If an operand (operand-1) precedes the BOOLh operator, its value is stored in the receiver-variable prior to performing the specified logical operation. For example, the statement:

```
200 A$ = C$ BOOL7 B$
```

first stores the current value of C$ into A$, and then not-AND's the value of B$ to A$. Again, the result of the operation is stored in A$. The contents of operand-1 and operand-2 are not affected by the operation.

In every case, the logical operation to be performed is identified by the hexadecimal digit following BOOL. A total of 16 logical operations are available (see Table P2-2). The hex digit used to identify each operation is a kind of mnemonic which represents the logical result of performing the operation on the following bit combinations:

receiver-variable:	1100
operand-2:	1010

For example, the hexdigit E identifies the OR operation. When 1100 is ORed with 1010, the result is 1110, or hexdigit E. Several commonly used BOOL operations are available as separate operators: BOOLE is equivalent to OR, BOOL6 to XOR, and BOOL8 to AND.

Table P2-2

BOOL digit	Logical Operation (Note: iff = if and only if)
0	null (bits always = 0; logical inverse of BOOLF)
1	not OR (1 iff corresponding bits of both arg 1 and arg 2=0) (NOR)
2	(1 iff corresponding bits of arg 2=1 and arg 1=0)
3	binary complement of arg 1 (1 iff bit of arg 1=0;) otherwise 0)
4	(1 iff corresponding bits of arg 2=0 and arg 1=1)
5	binary complement of arg 2 (1 iff bit of arg 2=0)
6	exclusive OR (1 iff corresponding bits of arg 1 and arg 2 are different) (XOR)
7	not AND (0 iff corresponding bits of both arg 1 and arg 2=1) (NAND)
8	AND (1 iff corresponding bits of both arg 1 and arg 2=1)
9	equivalence (1 iff corresponding bits are the same, i.e., both = 1 or both = 0)
A	arg 2 (identical to bits of arg 2)
B	arg 1 implies arg 2 (1 unless arg 1=1 and arg 2=0)
C	arg 1 (identical to bits of arg 1)
D	arg 2 implies arg 1 (1 unless arg 2=1 and arg 1=0)
E	OR (1 unless both corresponding bits = 0)
F	identity (bits always = 1; logical inverse of of BOOL(0)

BOOL6 is equivalent to XOR; BOOL8 is equivalent to AND; BOOLE is equivalent to OR.

Examples:

```
HEX(F000)=HEX(0F0F) BOOL1 HEX(0FF0)
HEX(F00F)=HEX(0F0F) BOOL5 HEX(0FF0)
HEX(FFFF)=HEX(0F0F) BOOLF HEX(0FF0)
```

CALL Statement

```
General Form:

CALL "name" [[ADDR](arg[,arg]...)]

where:
    "name" = 1-8 alphanumeric characters (including @, #, $)
           = SUB "name" of the SUB program being called.

     arg   = { expression          }
             { alpha-expression    }
             { array-designator    }
             { file-expression     }

Note:   Name must be enclosed in quotation marks.
```

CALL directs execution to the named subroutine, identified by a SUB statement, and passes any arguments to the subroutine program dummy arguments. The subroutine must be linked, using the LINKER utility, before the program is run. This can also be done when a program is compiled from the EDITOR.

The argument list in the CALL statement must correspond item-for-item with the argument list in the SUB statement, according to Tables P2-3 and P2-4.

Table P2-3

CALL argument	SUB argument
(alpha-)expression	scalar variable
matrix	matrix
vector	vector
file-expression	file-number

Table P2-4

CALL argument type	SUB argument type
alpha	alpha
floating-point	floating-point
integer	integer

A SUB statement with an argument list as follows:

```
100 SUB "HENRY" (ATLANTIS$, ELASMOBRANCH, JELLYFISH%(), #1)
```

must have arguments passed to it by a CALL statement in exactly the same order--in this case, alphanumeric scalar, floating-point variable, integer array-designator, file-expression. The arguments in the CALL statement do not have to be identical to those in the SUB statement, but each must correspond to the argument in the same position in the SUB statement's argument list. Thus, the following CALL statement is valid:

```
CALL "HENRY" (STR(C1$()), A(1), B%( ), #N)
```

Note that STR(C1$()) is used as a string since C1$() would be treated as an alpha array-designator.

Argument passing for the CALL statement proceeds as follows:

1. Values of file-expressions are passed to the SUB program to replace dummy file numbers (specifically, the UFB address is passed to the SUB program).

2. Pointers to the values of numeric scalar variables are passed to the SUB program.

For non-ADDR type -- Array and alphanumeric scalar descriptors are passed to the SUB routine, including pointers to the storage addresses, dimensions and lengths.

Since other numeric expressions and alpha expressions are not receivers, their values must be computed and stored in temporary locations, along with their lengths, if alphanumeric. Pointers (in the case of numeric expressions) or descriptors of the temporary values (in the case of alphanumeric expressions) are then passed to the SUB program.

Otherwise, execution proceeds as with arrays and receivers, except that returned values and lengths are effectively lost, since the locations are no longer accessible to the calling program.

For ADDR-type -- For all data types, pointers to the storage addresses only are passed; no dimensioning or length specifications are passed to the subroutine. (For numeric scalers and file-numbers this is identical to the non-ADDR type.)

Changed values are accessible as in non-ADDR type, except that array dimensions and lengths may be changed only within the subroutine, i.e., array dimensions and lengths will return to their original values after the subroutine returns to the calling program.

> **NOTE:**
>
> ADDR-type CALL is generally used only when the called subroutine is non-BASIC; otherwise, standard (non-ADDR) CALL's should be used.

Syntax examples:

```
100 CALL "ELIOT"(B,C$,D%)
200 PRINT "RETURNED"
300 STOP

100 DIM A$24
200 CALL "EXTRACT" ADDR("NA",A$)
300 PRINT A$
400 STOP

100 DIM LONG$100
200 CALL "123456" (LONG$)
300 PRINT LONG$
400 STOP
```

CLOSE Statement

```
General Form:

CLOSE ⎧file-expression⎫
      ⎨WS             ⎬
      ⎨CRT            ⎬
      ⎩PRINTER        ⎭
```

This statement closes a file that had been previously opened for I/O operations by an OPEN statement. If the file is subsequently re-opened in the program (by means of another OPEN statement), the file, library, and volume need not be respecified by the program or the user.

Attempting to close a file that has not been previously opened by an OPEN statement causes a nonrecoverable program error at run-time.

All files are closed at the start of the program; opened files should be closed before the end of the program.

CLOSE CRT allows the user to close the workstation. This is necessary if the user CALLs another program which attempts to OPEN the workstation. CLOSE CRT is equivalent to CLOSE WS.

CLOSE PRINTER is used to close the standard VS PRINT file selected by the SELECT PRINTER statement. Subsequent output to this device in the same run will be directed to another standard VS PRINT file. If the standard VS PRINT file is already closed, this statement has no effect.

Syntax examples:

```
100 CLOSE #1
300 CLOSE #A
500 CLOSE #LEN(A$)
700 CLOSE CRT
900 CLOSE PRINTER
```

PRINTER Programming Note

On program entry, the workstation is the default output device and the standard VS PRINT file is closed. If a SELECT PRINTER statement is executed, subsequent PRINT [USING] output is directed to the standard VS PRINT file. This standard file is implicitly opened the first time any output is generated by a PRINT [USING] statement following the execution of the SELECT PRINTER statement.

Several standard VS PRINT files may be created during a single program run. These multiple files may have different printline width specifications. The CLOSE PRINTER statement must be executed to signal the end of output to the open standard VS PRINT file, and the SELECT PRINTER option with a new width specified must be in effect for the next PRINT [USING] to be automatically routed to another standard VS PRINT file. Any attempt to alter the printline width while the standard VS PRINT file is open will produce a run-time error. To redirect output to the workstation a SELECT CRT or SELECT WS statement must be executed.

COM Statement

```
General Form:

COM com element [,com element]...

where:
                  ⎧numeric scalar variable                          ⎫
    com element = ⎨numeric array name (int [,int])                  ⎬
                  ⎨alpha scalar variable [length-integer]           ⎬
                  ⎩alpha array name (int [,int])[length-integer]    ⎭

              0 < length-integer ≤ 256
              0 < int ≤ 32767
```

The COM statement is also discussed in Subsection 6.5.4, Passing Values to External Subroutines.

The COM statement is a non-executable statement defining scalar variables or arrays to be used in common by several program segments.

This statement provides array definition identical to the DIM statement for array variables; a single COM statement can combine declarations of array variables (e.g., A(10), B(3,3)) and scalar variables (e.g., C2,D,X$).

Common variables must be defined before they are used. Therefore, it may be convenient to define the common variables at the beginning of the program.

If a particular set of common variables is to be used in each of several sequentially CALLed subprograms, the COM statement must be included in the main program and each subprogram in which they are used. All variables in the COM statements must be declared in the same order, and with the same dimensions and lengths, in each separately compiled module.

The COM statement can be used to set the maximum defined length of alphanumeric variables (assumed to be 16 if not specified). The length integer (≤256) following the alpha scalar (or alpha array) variable specifies the length of that alpha variable (or those array elements).

Syntax examples:

```
800 COM A(10),B(3,3),C2

200 COM C,D(4,14),E3,F(6),F1(5)

600 COM M1$,M$(2,4),X,Y

300 COM A$10,B$(2,2)32
```

CONVERT Statement

```
General Forms:

CONVERT alpha-expression TO numeric variable

[,DATA {GOTO } {line number     }]
       {GOSUB} {statement label }]

                    or

CONVERT numeric expression TO alpha-receiver, PIC (image)


where:
                        { # }    [ [ # ]                       ]
    image = [±][$]      { 0 }    [ [ 0 ]                 [ +  ]]
                        { * }    [ [ * ]...   [↑↑↑↑]     [ -  ]]
                        { B }    [ [ B ]                 [ ++ ]]
                        { / }    [ [ / ]                 [ -- ]]
                        { , }    [ [ , ]                       ]

where not both a leading and trailing sign may be used.
```

The CONVERT statement is used to convert alphanumeric representation of numeric data to internal numeric format, and vice versa. Two forms of the statement are provided.

Form 1: Alpha-to-Numeric Conversion

Form 1 of the CONVERT statement converts the number represented by ASCII characters in the alphanumeric expression to a numeric value and sets the numeric variable equal to that value. For example, if A$ = "1234", CONVERT A$ TO X sets X = 1234. An error will result (or the data exit will be taken) if the ASCII characters in the specified alphanumeric are not a legitimate BASIC representation of a number.

Alpha-to-numeric conversion is particularly useful when numeric data is read from a peripheral device in a record format that is not compatible with normal BASIC statements, or when a code conversion is first necessary. It can also be useful when it is desirable to validate keyed-in numeric data under program control. (Numeric data can be received in an alphanumeric variable, and tested with the NUM function before conversion to numeric format.) If the alpha-expression is entirely blank, an error will result (or the data exit will be taken).

Form 2: Numeric-to-Alpha Conversion

Form 2 of the CONVERT statement converts the numeric value of the specified expression to an ASCII character string according to the image specified. Numeric-to-alpha conversion is particularly useful when numeric data must be formatted in character format in records.

The image used with this form of CONVERT is used in the same way as a format-spec in an FMT statement, e.g., 100 CONVERT 10 to A$, PIC (####) Result: A$ = " 10"

Syntax examples:

Alpha to Numeric

```
    100 CONVERT A$ TO X
    200 CONVERT STR(A$,1,NUM(A$)) TO X(1)
```

Numeric to Alpha

```
    100 X = 12.195
    200 CONVERT X TO A$, PIC (000)
        (result:  A$ = "012")
    300 CONVERT X*2 TO A$, PIC (+##.##)
        (result:  A$ = "+24.39")
    400 CONVERT X TO STR(A$,3,8), PIC (-#.#++++)
        (result:  STR(A$,3,8) = " 1.2E+01")
    500 CONVERT X TO A$, PIC (0000.#####)
        (result:  A$ = "0012.19500")
```

COPY Statement

General Form:

COPY [-] alpha-expression TO [-] alpha-receiver

COPY transfers the alpha-expression to the alpha-receiver, one byte at a time, using the defined lengths of both.

If "-" is specified before the alpha-expression, the data is sent, starting from the rightmost byte of the expression, right-to-left. Similarly, if "-" is specified before the alpha- receiver, the data is received, starting from the rightmost byte of the receiver, right-to-left.

If "-" is not specified before the alpha-expression, the data is sent, starting from the leftmost byte of the expression, left-to-right. Similarly, if "-" is not specified before the alpha-receiver, the data is received, starting from the leftmost byte of the receiver, left-to-right.

Transfer stops when the receiver is filled, or the expression is exhausted (in which case the remainder of the receiver is filled with blanks).

NOTE:

If the alpha expression is a receiver, it is copied directly from its memory location; otherwise, the value of the alpha expression is stored into a temporary location and copied from there. Thus, COPYing a receiver onto itself can result in desirable or undesirable single-character propagation or other position-dependent results.

Syntax examples:

```
100 A$="CHART"
200 COPY A$ TO B$
    (result B$="CHART")
300 COPY -A$ TO B$
    (result B$="TRAHC")
```

COS Function

General Form:

```
COS(numeric expression)
```

The COS function returns a floating-point value that is the cosine of the numeric expression specified as its argument. The expression is considered to be in units of radians, degrees, or grads, depending on the trig mode specified by the most recently executed SELECT statement. If no SELECT statement has been executed in the program or subprogram, the default mode is radians.

DATA Statement

General Form:

$$\text{DATA} \begin{Bmatrix}\text{constant}\\ \text{literal}\end{Bmatrix} \left[,\begin{Bmatrix}\text{constant}\\ \text{literal}\end{Bmatrix}\right]\ldots$$

The DATA statement provides the values to be assigned to the variables in a READ statement. The READ and DATA statements thus provide a means of storing tables of constants within a program.

Each time a READ statement is executed in a program, the next sequential value(s) listed in the DATA statements are obtained and stored in the receivers listed in the READ statement. The values entered with the DATA statement must be in the order in which they are to be used; items in the DATA list are separated by commas. If several DATA statements occur in a program, they are used in order of statement number. Numeric variables in READ statements must reference numeric values; alphanumeric receivers must reference literals.

The RESTORE statement provides a means to reset the current DATA statement pointer and reuse the DATA statement values (see RESTORE).

Example:

```
100 FOR I=1 TO 5
200 READ W
300 PRINT W,W**2
400 NEXT I
500 DATA 5, 8.26, 14.8, -687, 22
```

```
Output: 5     25
        8.26  68.2276
        14.8  219.04
        -687  471969
        22    484
```

In the above example, the five values listed in the DATA statement are sequentially used by the READ statement and printed.

DATE Function

General Form:

DATE

DATE returns a 6-character string giving the current date in the form YYMMDD. The DATE function takes no arguments.

Example:

```
100 A$=DATE
200 PRINT STR(A$,3,2);"/";STR(A$,5,2);"/";STR(A$,1,2)
300 PRINT STR(DATE,3,2);"/";STR(DATE,5,2);"/";STR(DATE1,2)

Output:  06/15/79
         06/15/79
```

DEF Statement

```
General Form:

DEF function-name[%](v) = numeric expression

where:

    function-name = any sequence of up to 64 letters, digits,
                    and underscores provided that the first
                    character is a letter, and the name is not
                    a VS BASIC reserved word,

                v = the dummy variable, a numeric scalar variable.

    If % is present, the function will return an integer value.
```

The DEF statement is also discussed in Subsection 4.4.2, User-Defined Functions.

The define statement, DEF, enables the programmer to define a single-valued numeric function within the program. Once defined, this function can be used in expressions in any other part of the program. The function provides one dummy variable whose value is supplied when the function is referenced. Defined functions can reference other defined functions, but recursion is not allowed (i.e., a function cannot refer to itself, nor can a function refer to another function which refers to the first). The following program illustrates how DEF is used.

Example:

```
100 X=3
200 DEF OBFUSCATION(Z) = Z**2-Z
300 PRINT X + OBFUSCATION(2*X)
400 END

Output:  33
```

Processing of OBFUSCATION(2*X) in this example proceeds in the following order:

1. The expression specified as the argument of the function OBFUSCATION (in this case, 2*X) is evaluated. In this case, the value of the argument is (2*X=6).

2. The dummy variable in the function definition (in this case, Z in line 200) is temporarily assigned the value of the argument (in this case, 6).

3. The expression to the right of the equals sign in the function definition (line 200) is evaluated given the assignment just performed, and the value returned to the statement which invoked the function. In this case, (6↑2 - 6)=30 is returned to the PRINT statement (line 400), which adds the value of X (3, in this case), and prints the result (33).

A user-defined function may be invoked from anywhere in a program.

The following restrictions apply to definitions of functions:

1. A DEF function may not refer to itself; for example,

   ```
   DEF APPLE(MY_EYE) = MY_EYE + APPLE(MY_EYE)
   ```

   is illegal.

2. Two DEF functions may not refer to each other. For example, the following combination of statements is illegal.

   ```
   DEF ARTICHOKE(X) = BANANA(X)
   DEF BANANA(X) = ARTICHOKE(X)
   ```

Neither of the above restrictions is checked for during compilation, but both will cause endless loops resulting in "stack overflow" during execution.

The dummy scalar variable in the DEF statement can have a name identical to that of a variable used elsewhere in the program or in other DEF statements; current values of the variables are not affected during function evaluation. DEF statements may also use other variables, whose current values at calling time are used.

Syntax examples:

```
600 DEF JAGUAR(C) = (3*A) - 8*C + LION(2-A)
700 DEF LION(A) = (3*A) - 9/C
800 DEF TIGER(C) = LION(C) * JAGUAR(2)
```

DEF FN' Statement

```
General Form:

DEF FN' int  [(receiver[,receiver]...)]
             [literal[,literal]...]

where:
    int =   {1 to 32 for program function key entries }
            {0 to 255 for internal program references }
```

The DEF FN' statement has two purposes:

1. To define a literal to be supplied when a Program Function (PF) key is used for keyboard text entry.

2. To define Program Function key or program entry points for subroutines with argument passing capability.

Keyboard Text Entry Definition

To be used for keyboard entry, the integer in the DEF FN' statement must be from 1 to 32, representing the number of a Program Function key (PF key). When the corresponding PF key is pressed while execution is halted by an INPUT or STOP statement, the user's literal(s) is displayed and becomes part of the currently entered text line.

The literal may be represented by a character string in quotes, a HEX function or a combination of those elements.

NOTE:

The Program Function keys can be defined to output characters that do not appear on the keyboard by using HEX literals to specify the codes for these characters.

Examples:

```
100 DEF FN'31 "April is the cruelest month."
200 DEF FN'02 HEX(94); HEX(22);"Mistah Kurtz - he dead.";HEX(22)
```

Pressing PF 31 at a STOP or INPUT will cause "April is the cruelest month." to be displayed, while pressing PF 2 will cause "Mistah Kurtz - he dead." to appear, blinking and protected because of the HEX(94). The quotation marks are produced by HEX(22), which is an example of how it is possible to display characters which otherwise would be difficult to display.

Marked Subroutine Entry Definition

The DEF FN' statement, followed by an integer and an optional receiver list enclosed in parentheses, indicates the beginning of a marked subroutine. (See also Subsections 6.4.2, GOSUB' Subroutines and 6.4.3, Program Function Keys, for a discussion of marked subroutines.) The subroutine may be entered from the program via a GOSUB' statement or from the keyboard by pressing the appropriate Program Function key while execution is halted by an INPUT or STOP statement. If subroutine entry is to be made via a GOSUB' statement, the integer in the DEF FN' statement can be any integer from 0 to 255; if the subroutine entry is to be made from a Program Function key, the integer can be from 1 to 32. When a Program Function key is pressed or a GOSUB' statement is executed, the execution of the BASIC program transfers to the DEF FN' statement with an integer corresponding to the number of the Program Function key or the integer in the GOSUB' statement (i.e., if Program Function key 2 is pressed, execution branches to the DEF FN'2 statement).

When a RETURN statement is encountered in the subroutine, control is passed to the program statement immediately following the last executed GOSUB' statement, or back to the INPUT or STOP statement if entry was made by depressing a Program Function key.

Repeated subroutine calls executed without RETURN or RETURN CLEAR statements may cause memory overflow. (See RETURN and RETURN CLEAR.)

The DEF FN' statement may optionally include a receiver list. The receivers in the list receive the values of arguments being passed to the subroutine.

In a GOSUB' subroutine call made internally from the program, arguments are listed (enclosed in parentheses and separated by commas) in the GOSUB' statement. If the number of arguments to be passed is not equal to the number of receivers in the list, a compilation error results.

Example:

```
      100 GOSUB'2 (1.2,3+2 * X, "JOHN")
      .
      .
      200 STOP
      300 DEF FN'2 (A,B(3),C$)
      .
      .
      400 RETURN

      Result:  STOP 1.2, 3.24, "JOHN" (now press PF Key 2)
```

For Program Function key entry to a subroutine, arguments are passed by keying them in, separated by commas, immediately before the program function key is depressed. (See INPUT and STOP.) If the wrong number or type of data is given, the entries will be refused, the cursor will return to the beginning of the field, and the program will wait for further operator action.

The DEF FN' statement need not specify a receiver list. In some cases it may be more convenient to request data from a keyboard in a prompted fashion.

Example:

```
      100 DEF FN'4
      200 INPUT "RATE",R
      300 C = 100 * R - 50
      400 PRINT "COST=";C
      500 RETURN
```

When a DEF FN' subroutine is executed via keyboard Program Function keys while the system is awaiting data to be entered into an INPUT statement, or in STOP mode, the INPUT or STOP statement will be repeated in its entirety, upon return from the subroutine.

Example:

```
      100 INPUT "ENTER AMOUNT",A
          .
          .
          .
      200 DEF FN'1
      210 INPUT "ENTER NEW RATE",R
      220 RETURN
```

```
Display:    ENTER AMOUNT?
            (Press PF Key 1)
            ENTER NEW RATE? 7.5
            ENTER AMOUNT?
```

DEF FN' subroutines may be nested (i.e., call other subroutines from within a subroutine). A RETURN statement encountered in a nested subroutine will return execution to the subroutine which called the nested subroutine.

DELETE Statement

```
General Form:

DELETE file-expression
```

The DELETE command deletes the last record read, which must have been read with the HOLD option. It is only valid for INDEXED files; CONSEC records cannot be deleted.

See also the description of the HOLD option under the READ Disk File statement.

DIM Statement

```
General Form:

DIM dim-elt [,dim-elt]...

where:
    dim-elt = { numeric array name     (int1[,int2])         }
              { alpha array name       (int1[,int2])[int3]   }
              { alpha scalar variable [int3]                 }

       int1 = row dimension,    1≤int1≤32767
       int2 = column dimension, 1≤int2≤32767
       int3 = string length,    1≤int3≤256
```

The DIM statement reserves space for arrays and sets the length for alpha scalars or array variables. (Use of the DIM statement is also discussed in Subsection 3.5.2, Dimensioning an Array.)

The DIM statement must appear before use of any of the dimensioned elements.

If not dimensioned in a DIM statement, the following defaults hold:

1. The string length of alpha scalar or array variables defaults to 16. This is also true if int3 is omitted in a DIM statement.

2. Arrays default to 10-by-10 matrices.

3. Arrays or variables dimensioned in a COM statement may not be respecified in a DIM. (See COM statement.) A variable or array may occur in only one DIM or COM in each program or subprogram.

Arrays may be redimensioned by using [MAT] REDIM.

Syntax examples:

```
100 DIM A$100
200 DIM A$(4,4),B$(12,12)20,B(3,7)
300 DIM A(10),B$(20)10
```

Note that in a DIM statement, DIM must be the first word of the statement; if DIM is used in any other way, it is interpreted as referring to the DIM function.

DIM Function

General Form:

$$\text{DIM (array-designator, } \begin{Bmatrix} 1 \\ 2 \end{Bmatrix} \text{)}$$

where:
- {1} = corresponds to row dimension
- {2} = corresponds to column dimension

The DIM function returns, as an integer value, the current row (1) or column (2) dimension of the specified array. The column dimension of a vector is 1%.

NOTE:

The defined length of an alpha scalar or array variable may be obtained using LEN(STR(variable)).

Examples:

```
100 A=DIM(A(),1)
200 B=DIM(A(),2)
```

DISPLAY Statement

```
General Form:


DISPLAY   list    [,list]...

where:

    list  =  ⎧COL (int)                              ⎫
             ⎪AT(exp2, exp3)                         ⎪
             ⎨numeric expression [,PIC(image)]       ⎬
             ⎪alpha-exp [,CH (int)]                  ⎪
             ⎩BELL                                   ⎭

    image = a valid numeric image, as in FMT.
    int   = an int specifying the length of the (alpha) field.
```

DISPLAY allows the output of numeric and alphanumeric data values via the workstation in a field-oriented manner, using the supplied formatting information. (See Section 7.6, The Display Statement for a detailed discussion.) Both single values and arrays may be output.

DISPLAY works in generally the same way as ACCEPT, with the following exceptions:

1. Values are written only; no new values are accepted. (Thus there are no PF key clauses or FAC characters.)

2. Pseudoblanks are not used.

Otherwise, see ACCEPT. The screen is cleared prior to DISPLAY, and a STOP statement should be used in order to halt execution for viewing (if desired) following DISPLAY.

See Chapter 7 for more information on screen I/O.

Examples:

```
100 DISPLAY COL(10),A$,CH(20),AT(20,20),A,PIC(##.##)
200 DISPLAY B$,BELL
```

EJECT Compiler Directive

```
General Form:

EJECT
```

EJECT is a compiler directive (see Section 2.4, Subsection 2.4.2, Compiler Directives). The EJECT statement, which must be the only statement on a line, causes the compiler to skip to the top of the next page of the source listing and print the most recently specified title at the beginning of the page.

END Statement

```
General Form:

END [expression]
```

This statement is required to terminate the program prior to its physical end or to pass a program-supplied return code to the operating system. It may be used anywhere and any number of times in the program. It is not required at the physical end of the program where an implied END is automatically generated.

When the END statement is encountered, program execution terminates or, if in a subroutine, execution returns to the calling program. If END is followed by an expression, the value of the expression (truncated if not an integer) is passed to the operating system as a return code. If 'expression' is omitted, the return code is 0; e.g., 100 END 999 END A

The second example passes the current (truncated) value of A to the system as a return code.

Return codes are often useful in writing procedures. (See the VS Programmer's Introduction for a discussion of procedures and the use of return codes.)

EXP Function

General Form:

```
EXP(numeric expression)
```

The EXP (exponential) function returns a floating-point value equal to the natural constant "e" (the base of natural logarithms; e = 2.71828182845904) raised to the power given by the value of the argument. EXP is the inverse function of LOG.

Examples:

```
100 A = EXP(1)
200 B = EXP(73)
300 PRINT A, B

Result:  2.718281828       9744803446
```

FMT Statement

```
General form:

FMT  form-spec  [ ,  form-spec  ]...

where:

    form-spec = ⎧[rep-int*]data-spec⎫
                ⎨[rep-int*]literal   ⎬
                ⎩control-spec        ⎭


    rep-int   = int specifying the number of times to
                repeat the data-spec or literal.
```

FMT is a non-executable statement used to format data values for PRINT and disk I/O statements. (See also Section 7.4, The USING Clause and Format Control Statements, and 8.4, The File I/O Statements, for discussions of the use of the FMT statement.) The FMT statement and the FORM statement are synonymous and may be used interchangeably. They may be used wherever Image(%) is allowed, subject to the following restrictions:

1. BI, FL, and PD are not displayable formats, and thus are legal <u>only</u> for disk I/O statements.

2. For PRINTUSING, the FMT statement may be re-used for long argument lists. This is exactly like Image, and is described in the PRINTUSING section.

Control-Spec

1. <u>XX [(int)]</u> -- Skip int positions (input) or write n blanks (output). Omitted int=1.

2. <u>COL (int) or POS (int)</u> -- Next form-spec to start at position int in record or output line. (For disk I/O, int $\leq$ record size. For PRINTUSING, COL>80 or current printer width causes the next form-spec to begin at column 1 of the next line.)

3. <u>TAB (int)</u> -- Like COL, but all skipped-over characters are set to blank.

4. <u>SKIP [(int)]</u> -- Skip int lines (default=1). Like PRINT SKIP. (<u>Not</u> for disk I/O.)

Data-spec (Note: w and d are int constants.)

1. CH(w) -- Character data, w bytes.

2. BI[(w)] -- Binary internal format, w bytes. $1 \le w \le 4$, default=4.

3. FL[(w)] -- Floating-point internal format, w bytes w=4 or 8, default=8.

4. PD(w[,d]) -- VS packed decimal, w digits, d digits to the right of the (implied) decimal point (default d=0). Number of bytes required is 1+INT(w/2).

5. $$\text{PIC}([\pm][\$]\left\{\begin{matrix}\#\\0\\*\\B\\/\\,\end{matrix}\right\}\ldots \quad . \left[\left[\begin{matrix}\#\\0\\*\\B\\/\\,\end{matrix}\right]\ldots[\uparrow\uparrow\uparrow\uparrow]\quad\left[\begin{matrix}+\\-\\++\\--\end{matrix}\right]\right])$$

Editing Characters

#	Digit position - blank if leading zero.
.	Decimal point.
↑↑↑↑	Exponent E±xx for exponential output. If present, the digit positions will be filled with significant digits (no leading zeros) and the exponent scaled accordingly.
*	Replace leading 0 with *.
0	Retain leading 0.
,	If right of a numeric digit, insert ',' ; otherwise, blank.
/	If right of a numeric digit, insert '/' ; otherwise, blank.
B	Insert blank.

Sign Trailing	+	'+' > 0, '-' if < 0
	-	blank if > 0, '-' if < 0
	++	2 blanks if > 0, 'CR' if < 0
	--	2 blanks if > 0, 'DB' if < 0

1. A leading sign and a trailing sign cannot both be specified.

2. If no signs are present, the absolute value of the number is printed.

Sign Leading	+	'+' if > 0, '-' if < 0
	-	blank if > 0, '-' if < 0
	$	'$' precedes the number

(The above three characters float to the leftmost nonzero digit location.)

Examples:

```
100 FMT PIC(##.##++++)
200 FMT SKIP(10),CH(5),SKIP(-5),COL(20),PIC($**.##)
```

NOTE:

The FMT Statement always extends to the end of the line on which it occurs. It cannot be terminated by use of a colon (:) as described in Section 2.3.2, Multiple Statement Lines.

FOR Statement

General Form:

```
FOR numeric scalar variable = expl TO exp2 [STEP exp]
```

The FOR and NEXT statements are used to specify a loop. The FOR statement marks the beginning of the loop and defines the loop parameters. The NEXT statement marks the end of the loop. The program lines in the range of the FOR statement are executed repeatedly, beginning with variable = expl, and thereafter incremented by the STEP expression value until the variable value exceeds the value of exp2.

The three expressions may take on any value. If STEP is omitted, 1 is assumed. STEP and exp2 are evaluated only once; if STEP is 0 or has the wrong sign, the loop is executed only once.

After termination of the loop, the variable has the last value used, i.e., without the final increment. There are no restrictions on branching in or out of the loop, provided that a NEXT without an open FOR is not encountered; this event will cause an error.

NOTE:

If the loop variable is an integer variable, expl, exp2 and the step exp will be truncated to integers and all loop calculations will be integer type.

Example:

```
100 FOR A=1 TO 10 STEP 3
200 PRINT A
300 NEXT A
```

Result:

```
1

4

7

10
```

FORM Statement

```
General form:

FORM  form-spec  [ ,  form-spec  ]...

where:

    form-spec = ⎧[rep-int*]data-spec ⎫
                ⎨[rep-int*]literal   ⎬
                ⎩ control-spec       ⎭

    rep-int   = int specifying the number of times to
                repeat the data-spec or literal.
```

Synonymous with FMT.

FS Function

General Form:

```
FS (file expression)
```

The FS function returns the file status for the most recent I/O operation on the specified file, as an alpha value two characters long. FS can assume any of the following values:

CONSEC, TAPE, and PRINTER file I/O

'00'	Successful I/O operation
'10'	End-of-file encountered
'23'	Invalid record number
'30'	Hardware error
'34'	No more room in the file
'95'	Invalid function or function sequence
'97'	Invalid record length

INDEXED file I/O

'00'	Successful I/O operation
'10'	End-of-file encountered
'21'	Key out of sequence (WRITE statement in OUTPUT mode only)
'22'	Duplicate key
'23'	No record found matching specified key
'24'	Supplied key exceeds any key in the file (INPUT, I/O, or SHARED mode)
'34'	No more room in the file (OUTPUT or EXTEND mode)
'30'	Hardware error
'95'	Invalid function or function sequence
'97'	Invalid record length

SHARED MODE I/O ERRORS*

'80'	Invalid Key area (START, READ KEYED)
'81'	Invalid READ NODATA
'82'	Label update error
'83'	Sharing task was terminated
'84'	Invalid record size/record area (Record size > 2048)

*Not normally encountered by BASIC user

GET Statement

```
General Form:

GET {file-exp } [[,] USING {line number    }], arg [,arg]...
    {alpha-exp}             {statement label}

    [, DATA {GOTO } {line number    }]
    [       {GOSUB} {statement label}]

 where:
    arg ={receiver        }
         {array-designator}
```

GET allows extraction of data from the record area in a file or from an alpha-expression USING the referenced Image (%) or FMT statement, or using standard format.

Data in the record area referenced by the file-expression are those read with the last READ statement; these data are available to GET until overwritten by another READ from the same file, or by a PUT, WRITE, or REWRITE for that file.

The DATA exit is taken if data conversion fails (e.g., character string moved to numeric variable, alpha-expression too short to fill all the args, etc.).

Syntax examples:

```
100 GET #A USING 300,B,DATA GOTO 500
300 FMT PIC(####)
```

NOTE:

GET may be used to convert numeric data from internal formats used by COBOL programs to BASIC numeric data format. See Appendix D for information on numeric data compatability between BASIC and COBOL.

GOSUB Statement

General Form:

$$\text{GOSUB}\begin{Bmatrix}\text{line number}\\ \text{statement label}\end{Bmatrix}$$

The GOSUB statement is used to transfer program execution to the first program line of a subroutine. (The use of the GOSUB statement is also discussed in Subsection 6.4.1, GOSUB Subroutines.) The program line may be any BASIC statement, including a REM statement or a statement label line. The logical end of the subroutine is a RETURN or RETURN CLEAR statement. A RETURN statement directs execution to the statement following the last executed GOSUB; a RETURN CLEAR statement clears the subroutine information but causes no branch.

```
120 X = 20:GOSUB 200:PRINT X
125
130 GOSUB TEST
.
.
.
190 TEST:
200 REM SUBROUTINE BEGINS
.
.
.
210 RETURN:REM SUBROUTINE ENDS
```

The GOSUB statement may be used to perform a subroutine within a subroutine; this technique is called "nesting" of subroutines.

Repeated entries to subroutines without executing a RETURN or RETURN CLEAR should not be made. Failure to execute a RETURN or RETURN CLEAR causes return information to be accumulated in a table which eventually will cause a memory stack overflow error.

GOSUB' Statement

```
General Form:

GOSUB'int[(arg[,arg]...]

where:
    0<int<256
          ⎧expression      ⎫
    arg = ⎩alpha expression⎭
```

The GOSUB' statement specifies a transfer to a marked subroutine rather than to a particular program line, as with the GOSUB statement. (The use of the GOSUB' statement is also discussed in Subsection 6.4.2, GOSUB' Subroutines.) A subroutine is marked by a DEF FN' statement. When a GOSUB' statement is executed, program execution transfers to the DEFFN' statement having an integer identical to that of the GOSUB' statement (i.e., GOSUB'6 would transfer execution to the DEF FN'6 statement). Subroutine execution continues until a subroutine RETURN or RETURN CLEAR statement is executed. The rules applying to GOSUB usage also apply to the GOSUB' statement. Unlike a normal GOSUB, however, a GOSUB' statement can contain arguments whose values can be passed to variables in the marked subroutine.

The values of the expressions, literal strings, or alphanumeric variables are passed to the variables in the DEF FN' statement left to right. Elements of arrays must be explicitly referenced (i.e., they cannot be referenced by the array-designator or array name alone). The arguments of the GOSUB' must be passed to variables of the same type (i.e., alpha expressions must be passed to alpha variables, and numeric expressions must be passed to numeric variables).

Repetitive entries to subroutines without executing a RETURN or RETURN CLEAR should not be made. Failure to execute a RETURN or RETURN CLEAR causes return information to accumulate in a table, which could eventually cause a stack overflow error.

Examples:

```
100 GOSUB'7
150 END
200 DEF FN'7:SELECT PRINTER (80)
210 RETURN
```

```
100 GOSUB'12 ("JOHN",12.4,3*X+Y)
200 END
300 DEF FN'12(A$,B,C(2))
400 PRINT A$,B,C(2)
500 RETURN
```

GOTO Statement

General Form:

$$\text{GOTO} \begin{Bmatrix} \text{line number} \\ \text{statement label} \end{Bmatrix}$$

This statement transfers execution to the specified line number or statement label; execution continues at the specified line statement.

Example:

```
100 J=25
200 K=15
300 GOTO TEST
400 Z=J+K+L+M
500 PRINT Z,Z/4
600 END
650 TEST
700 L=80
800 M=16
900 GOTO 400

Output: 136       34
```

HEX Function

```
General Form:

HEX(hh[hh]...)

where:
    h = hexdigit (0 to 9 or A to F)
```

The hexadecimal function, HEX, is a form of literal string that enables any 8-bit code to be used in a BASIC program. Each character in the literal string is represented by two hexadecimal digits. If the HEX function contains an odd number of hexdigits or if it contains any characters other than hexdigits, an error results.

Syntax examples:

```
100 A$=HEX(0C0A0A)
200 IF A$ > HEX(7F) THEN 100
300 PRINT HEX(8001);"TITLE"
```

HEXPACK Statement

General Form:

```
HEXPACK alpha-receiver FROM alpha-expression

      [, DATA {GOTO } {line number    }]
      [       {GOSUB} {statement label}]
```

The HEXPACK statement converts an ASCII character string which represents a string of hexadecimal digits into the binary equivalent of those hex digits. Hexadecimal digits entered from the keyboard may be entered as ASCII characters; they may then be converted from ASCII code to their true binary equivalent with HEXPACK. For example, the hex digit A has a binary value of 1010. However, this digit is represented by an ASCII character A, which has a binary value of 01000001. The HEXPACK statement can be used to convert the binary value of ASCII character A into the binary value of the hexadecimal digit A, and to store this value in the specified alpha-receiver.

The alpha-expression (actual length) contains the ASCII character string which represents a string of hexadecimal digits. Each pair of ASCII characters is converted to one byte of the corresponding binary value. Only certain ASCII characters constitute legal representations of hexadecimal digits. These include the characters 0-9 and A-F, as well as the special characters :, ;, <, =, >, and ?. These characters are converted to the following binary values:

ASCII Character	Binary Value
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	000
9	1001
A or :	1010
B or ;	1011
C or <	1100
D or =	1101
E or >	1110
F or ?	1111

If the alpha-expression contains any characters other than those listed above, including embedded spaces (i.e., any character which is not a legal representation in ASCII of a hexadecimal digit), an error occurs; if the DATA exit is specified, it is taken.

If the alpha-expression contains an odd number of legal hex digits, it is padded on the right with one hex zero.

The alpha-receiver receives the converted binary value. Since each pair of characters in the value of the alpha-expression is converted to a one-byte binary value in the alpha-receiver, the alpha-receiver should have at least half as many bytes (defined length) as the alpha-expression. If the alpha-receiver is too short to contain the entire converted binary value, an error occurs and program execution halts. If the alpha-receiver is longer than the converted binary value, the binary value is left-justified, and the remaining bytes of the alpha-receiver are not modified.

Example 1:

```
100  DIM P$2, U$4
200  INPUT "VALUE TO BE PACKED",U$
300  HEXPACK P$ FROM U$
400  PRINT HEXOF (P$)
```

Output:

```
VALUE TO BE PACKED?12C9

12C9
```

The availability of the special characters ":" (HEX (3A)) through "?" (HEX (3F)) to represent hex digits A-F (1010-1111) means that HEXPACK will recognize any ASCII code with a high-order "3" digit (hex 30 through hex 3F) as a legitimate representation of a hexadecimal digit. This fact makes it easy to transform any code into an acceptable representation of a hex digit, and hence to perform operations such as packing the low-order digits (low-order four bits) from a string of hexadecimal digits. The technique is illustrated in Example 2.

Example 2:

```
100  DIM P$2, V$4
200  V$ = HEX (01020C09)
300  V$ = OR ALL (HEX(30))
400  HEXPACK P$ FROM V$
500  PRINT HEXOF(P$)

Output:  12C9
```

Syntax examples:

```
HEXPACK A$ FROM B$
HEXPACK STR(A$,1,3) FROM STR(B$,7)
HEXPACK A$() FROM B$()
HEXPACK A$ FROM "3AFC282C"
```

HEXPRINT Statement

General Form:

$$\text{HEXPRINT} \begin{Bmatrix}\text{alpha variable}\\ \text{alpha array designator}\end{Bmatrix} \left[\left[\begin{Bmatrix},\\ ;\end{Bmatrix} \begin{Bmatrix}\text{alpha variable}\\ \text{alpha array designator}\end{Bmatrix}\right] \ldots \right] [;]$$

This statement prints the value of the alpha variable or the values of the alpha array in hexadecimal notation. The printing or display is done on the device currently selected for PRINT operations (see SELECT). The defined lengths of the alpha values are printed. Arrays are printed one element after another with no separation characters. A new line is started after the value(s) of each alpha variable (or array) in the argument list, unless the argument is followed by a semicolon. If the printed value of the argument exceeds one line on the workstation or printer, it will be continued on the next line or lines. Since the carriage width for PRINT operations can be set to any desired width by the SELECT statement, this could be used to format the output from long arguments.

Note that HEXPRINT X$, Y$, Z$ is the same as PRINT HEXOF (X$), HEXOF (Y$), HEXOF (Z$).

HEXUNPACK Statement

```
General Form:

HEXUNPACK alpha-expression TO alpha-receiver
```

The HEXUNPACK statement converts the binary value of an alpha-expression (defined length) to a string of ASCII characters representing the hexadecimal equivalent of that value. The resulting characters are stored in the alpha-receiver.

HEXUNPACK is the logical inverse of HEXPACK, with the exception that characters 3A-3F are not used; the characters produced are in the range 0-9 and A-F.

If the alpha-receiver is not at least twice as long as the alpha-expression (defined length), an error occurs. If it is longer, the result is left-justified and unused characters remain unchanged (as with HEXPACK).

Example:

```
100  DIM P$2, U$4
200  P$ = HEX (12C9)
300  HEXUNPACK P$ TO U$
400  PRINT U$

Output: 12C9
```

Syntax examples:

```
HEXUNPACK A$ TO B$
HEXUNPACK STR(A$, 5) TO STR(B$, 1, 4)
HEXUNPACK A$() TO B$()
```

IF...THEN...ELSE Statement

```
General Form:

IF relation THEN (line number  ) [ELSE  (line number  ) ]
                 (executable   ) [      (executable   ) ]
                 ( statement*  ) [      ( statement*  ) ]
                 ( statement   ) [      ( statement   ) ]
                 (      label  ) [      (      label  ) ]

                                          *except another IF

where:
    relation = (alpha exp operator alpha exp              )
               (numeric exp. operator numeric exp.        )
               (                                          )
               (                  (AND)                   )
               (     relation     (OR )relation           )
               (                  (XOR)                   )
               (                                          )
               (NOT relation                              )
               ((relation)                                )

                            (< )
                            (> )
    operator =              (<=)
                            (>=)
                            (<>)
                            (= )
```

The IF statement causes conditional transfer or statement execution. The following may occur, depending on the value of the relation:

1. Relation <u>true</u>:

   If "THEN line number (or statement label)" is specified, execution continues at the specified line number or statement label.

   If "THEN executable statement" is specified, the statement is executed. Program execution then continues at the next executable statement.

   In either case, the ELSE clause is ignored.

2. Relation false:

If the ELSE clause is not specified, execution continues at the next executable statement.

If the ELSE clause is specified, it is used like THEN in 1 and 2 above.

In either case, the THEN clause is ignored.

Two expressions are compared using standard numerical order; integers are converted to floating-point before being compared with floating-point values.

Two alpha-expressions are compared using their ASCII hexcodes, with the shorter expression right-padded with blanks (HEX(20)).

The hierarchy of execution of the relational expression is as follows:

1. Parentheses
2. <,<=,>,>=,<>,=
3. NOT
4. AND, OR, XOR
5. Left-to-right execution

NOTE:

Nested IF statements are not allowed.

Syntax examples:

```
100 IF A > .5 THEN 1000
200 IF A$>B$ AND B$>C$ THEN B=5 ELSE B=0
300 IF NOT A=B THEN 1000
400 IF E$<=F$ AND (NOT N>I) THEN 1000 ELSE 800
500 IF A>B THEN TEST ELSE NO_TEST
```

Image(%) Statement

```
General Form:

% ⎧character string    ⎫ ...
  ⎩format specification⎭

where:

   Character string = ⎧any character⎫       ...
                      ⎩except '#'   ⎭

   Format specification ⎡⎧=⎫                                                    ⎡ + ⎤⎤
                        ⎢⎨+⎬  [$]#...[,][#...[,]]...[.][#...][↑↑↑↑]          ⎢ - ⎥⎥
                        ⎢⎩-⎭                                                    ⎢++ ⎥⎥
                        ⎣                                                       ⎣-- ⎦⎦
```

The Image(%) statement is a non-executable statement used to format output from PRINTUSING, disk I/O and GET and PUT statements. One format specification is used per numeric or alpha value, left to right.

For alphanumeric values, the format specification is filled from left to right, regardless of the editing characters. The output value is right-padded with blanks or truncated to fit the format spec.

For numeric values, the editing characters in the format spec are interpreted depending upon the value to be formatted.

Format Characters

```
Leading   +   '+' if    0, '-' if    0
          -   blank if    0, '-' if    0
          $   '$' precedes the number
```

(The above three characters float to just before the leftmost nonzero digit location.)

```
#   digit position - blank if leading zero.
.   decimal point.
,   comma if at least 1 significant digit is positioned
    to the immediate left; otherwise blank.
    exponent E±xx for exponential output.  If
    present, the digit positions will be filled with
    significant digits (no leading zeros) and the
    exponent scaled accordingly.
```

Trailing	+	'+' if >0, '-' if <0
	-	blank if >0, '-' if <0
	++	2 blanks if >0, 'CR' if <0
	--	2 blanks if >0,'DB' if <0

NOTE:

1. If a leading sign is present, the trailing sign is ignored, i.e., it becomes (part of) the next character string.

2. If no signs are present, the absolute value of the number is printed.

1. Note that there must be at least a single '#' in a format specification, and that the output field width is always the same length as the format specification, whether the output is numeric or alphanumeric.

2. For numeric output:

   - Fractions are truncated.

   - If the format is insufficient for the integer part of the number, the format specification itself is output, with the correct leading sign, if the leading sign character is present.

3. If all format specifications are not used, everything up to the first unused format is used, including a final character string.

4. A trailing character string in an IMAGE statement is considered to extend to the last nonblank character.

5. A continued Image line is used up to the '!' character.

Syntax examples:

```
100 %FEAR IN A HANDFUL OF DUST +###,###,###.##

100 ACCEPT A,B,C
200 PRINTUSING 300, A,B,C
300 %$##,###.##++ ###.###-- -###.##↑↑↑↑
400 STOP
500 GOTO 100
```

INIT Statement

General Form:

```
INIT (alpha-exp) alpha-receiver [,alpha-receiver]...
```

The INIT statement initializes the specified alphanumeric receivers. Each character in the defined length of the alpha-receiver(s) is set equal to the first character of the alpha-expression. For example,

```
INIT("?")A$,B$,M$()
```

sets both alpha variables A$ and B$ and alpha array variable M$() to contain all question mark characters.

INPUT Statement

```
General Form:

INPUT [literal,] receiver [,receiver]...
```

This statement allows the user to supply data during the execution of a program. If the user wants to supply the values for A and B while running the program,

```
400 INPUT A,B
        or
400 INPUT "VALUE OF A,B",A,B
```

would be entered before the first program line that requires either of these values (A,B). When the system encounters this INPUT statement, it outputs the message VALUE OF A,B, followed by a question mark (?), and waits for the user to supply the two numbers. Once the values have been supplied, program execution continues. The program assigns values left to right, one at a time. The device used for inputting data is the workstation.

Each value must be entered in the order in which it is listed in the INPUT statement and values entered must be compatible with receivers in the INPUT statement. If several values are entered, they must be separated by commas or entered on separate lines. As many lines as necessary may be used to enter the required INPUT data. To include leading blanks or commas as part of an alpha value, enclose the value in double or single quotes (" or '), for example, "BOSTON, MASS.".

Variables in the INPUT list which the user does not wish to change may be skipped over by entering a null value, i.e., a comma not immediately preceded by a data item. For example,

Program:	VALUE OF A,B,C,D?
User :	4.3,2.0,,3.5
Result :	Variable C will not be changed; A,B, and D get new values.

A user may terminate an input sequence without supplying any additional input values by simply keying ENTER with no other information preceding it on the line. This causes the program to immediately proceed to the next program statement. The INPUT list receivers which have not received values remain unchanged.

When inputting alphanumeric data, literal strings need not be enclosed in quotes. However, leading blanks are ignored and commas act as string terminators. (This applies to subroutine parameters also -- see Subsection 7.5.3, PF Key Usage and Program Branching.)

Example 1:

```
     100 INPUT X

     Output: ?12.2 (ENTER)
             (underlined portion supplied by user)
```

Example 2:

```
     200 INPUT "MORE INFORMATION",A$
     300 IF A$="NO" THEN END
     400 INPUT "ADDRESS",B$
     500 GOTO 200

     Output: MORE INFORMATION? YES (ENTER)
             ADDRESS? BOSTON, MASS (ENTER)
             MORE INFORMATION? NO (ENTER)
```

Program Function Keys in Input Mode

Program Function (PF) Keys may be used in conjunction with INPUT. If the PF key has been defined for text entry (see DEF FN') and an INPUT statement is executed, pressing the PF key causes the character string in the DEF FN' statement to be displayed on the CRT. The displayed value is stored in the variable which occurs in the INPUT statement when the ENTER key is pressed. For example:

```
     100 DEF FN'01"COLOR T.V."
     200 INPUT A$

     Result: ?
                 (Now, pressing PF 1
                  will cause "COLOR T.V." to appear on the CRT.)
         ?COLOR T.V._
                    (CRT Cursor)
```

If the PF key is defined to call a marked subroutine, (see DEF FN') and the system is awaiting input, pressing the PF key will cause the specified subroutine to be executed. No assignment occurs, and the values keyed before hitting the PF key are ignored, unless the subroutine has an argument list. If so, as many values as are required are taken, starting from the leftmost value keyed; those left over are ignored. The workstation alarm sounds if there are too few values or if those values do not correspond correctly to the receivers in the GOSUB' argument list. An illegal PF key also causes an alarm. When the RETURN statement is encountered, control returns to the INPUT statement and the INPUT statement will be executed again. Repeated subroutine entries via PF keys should not be made unless a RETURN or RETURN CLEAR statement is executed; otherwise return information accumulates in a table and eventually causes a stack overflow error.

Example:

The program below enters and stores a series of numbers. When PF 02 is pressed, they are totaled and printed.

```
100 DIM A(30)
200 N=1
300 INPUT "AMOUNT",A(N)
400 N=N+1:GOTO 300
500 DEFFN'02
600 T=0
700 FOR I=1 TO N
800 T=T+A(I)
900 NEXT I
1000 PRINT "TOTAL=";T
1100 N=1
1200 RETURN
```

```
Output: AMOUNT? 7  (ENTER)
        AMOUNT? 5  (ENTER)
        AMOUNT? 11 (ENTER)
        AMOUNT? (Depress PF 2)
        TOTAL = 23
        AMOUNT?
```

INT Function

General Form:

```
INT(numeric expression)
```

The INT (integer) function returns an integer value that is the greatest integer less than or equal to the value of the numeric expression specified as the argument.

Examples:

```
INT( 1.5) =  1
INT(-1.5) = -2
```

KEY Function

General Form:

```
KEY (file expression [,exp])
```

KEY returns the primary key (or an alternate key) of the last record read from the specified file. If exp is 0 or omitted, the primary key is returned. Otherwise, the alternate key with key number = exp (from SELECT) is returned. (For alternate-indexed files only.)

The length of the result is the (primary or alternate) key length as specified in SELECT.

KEY may also be used as a receiver to set the (primary or alternate) key field in the record prior to WRITE or REWRITE.

LEN Function

General Form:

LEN (alpha-expression)

LEN determines the actual length, in bytes, of the alpha-expression. It can be used wherever a numeric expression is permitted. The result of LEN is an integer value.

Example:

```
     100 A$ = "ABCD"
     200 PRINT LEN (A$)
```

These program lines print the value 4 at execution time.

Example:

```
     300 X = LEN(A$)+2
```

Combined with lines 100 and 200 above, this line assigns the value 6 to X at execution time.

Example:

```
     100 A$ = "ABCD"
     200 PRINT LEN(STR(A$,2))
```

These lines give the value 15 at execution time. Since A$ is not explicitly dimensioned, the default value for its length is 16 bytes. The STR function extracts the bytes from A$, starting at the second byte, to its end. The length of such a value is 15.

Example:

```
     100 DIM A$64
     200 A$ = "ABCD"
     300 PRINT LEN(STR(A$,POS(A$=HEX(20))))
```

These lines give the value 60 at execution time. The length of the alpha scalar is initially 64; the value of the POS function is first determined, giving the position of the first blank character in A$ equal to 5. The STR function then extracts the number of bytes from the first blank character to the end of the scalar.

LET Statement

```
General Form:

[LET] numeric variable [,numeric variable]... = numeric expression
                                  or
[LET] alpha-receiver [,alpha-receiver]... =  alpha-expression
                                  or
[LET] alpha-receiver = logical expression
```

The LET statement evaluates the expression following the equal sign and assigns the result to the receiver(s) specified preceding the equal sign. If more than one receiver appears before the equal sign, they must be separated by commas. If the right-hand side of the statement is a logical expression (see Section 5.7, Logical Expressions), then only one receiver may appear on the left.

An error results if a numeric value is assigned to an alphanumeric receiver, or if an alphanumeric value is assigned to a numeric variable.

Examples:

```
400 LET X(3),Z,Y=P+15/2+SIN(P-2.0)


500 LET J=3


(In this example, LET is assumed)
100 X=A*E-Z*Y
200 A$=B$
300 C$,D$(2)="ABCDE"


100 C$ = 'ABCDE'
200 A$ = "123456"
300 D$ = STR(A$,2)
400 E$ = HEX(41)
500 PRINT A$,C$,D$,E$
```

This routine produces the following output at execution time:

```
     123456       ABCDE       23456       A
```

The execution of

```
          [LET] rec1, rec2,..., recn = value
```

is equivalent to

```
     [LET] recn   = value

     [LET] recn-1 = value
                 .
                 .
                 .
     [LET] rec1  = value
```

for both alpha and numeric assignment. Assignment is right-to-left.

LGT Function

```
General Form:

LGT(numeric expression)
```

The LGT function returns a floating-point value equal to the common (base 10) logarithm of the numeric expression specified as the argument.

Example:

```
LGT (100) = 2
```

LOG Function

```
General Form:

LOG(numeric expression)
```

The LOG function returns a floating-point value equal to the natural logarithm (base "e") of the argument. LOG is the inverse function of EXP.

Example:

```
LOG (10) = 2.3025
```

MASK Function

General Form:

MASK (file expression)

MASK returns the alternate key access mask (alternate indexed file) for the last record read from the specified file. The result is a 2-byte (16 bit) alpha value whose bits (left to right) correspond to available alternate keys (1-16). Bits which are "on" (binary 1) specify that the record may be accessed, via a READ statement with a key clause, by those alternate key paths.

The MASK function may also be used as a receiver to set the alternate key mask for a record prior to a WRITE or REWRITE statement.

Examples:

```
A$=MASK(#1)
MASK(#2)=HEX(FF00)
```

MAT + (MAT addition) Statement

```
General Form:

MAT c = a + b

where:
    c, a, and b are numeric array names.
```

This statement adds two matrices or vectors of the same dimension. The sum is stored in array c. Any two or all of a, b, and c may be the same array. Array c is implicitly redimensioned to have the same dimensions as arrays a and b.

An error occurs and execution is terminated if the dimensions of a and b are not the same.

Example 1:

```
100 DIM A(5,5),D(5,5),E(7),F(5),G(5)
200 MAT A=A+D
300 MAT E=F+G
400 MAT A=A+A
```

Example 2:

The program provided adds the corresponding elements of the 3-by-3 arrays D and E, to give the new array F. Array F is automatically redimensioned as a 3-by-3 array.

```
100 DIM D(3,3),E(3,3),F(5,2)
200 PRINT "ENTER ELEMENTS OF ARRAY D"
300 MAT INPUT D
400 PRINT "ENTER ELEMENTS OF ARRAY E"
500 MAT INPUT E
600 MAT F=D+E
700 PRINT "ELEMENTS OF ARRAY F":PRINT
800 MAT PRINT F;
```

Let D= $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$ E= $\begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix}$

When the program is executed, array F is displayed:

```
ELEMENTS OF ARRAY F

 4   4   4
 4   4   4
 5   5   5
```

MAT ASORT/DSORT Statement

General Form:

$$\text{MAT numeric array name1} = \left\{\begin{matrix}\text{ASORT}\\ \text{DSORT}\end{matrix}\right\}\text{(numeric array name2)}$$

$$\text{MAT alpha array name1} = \left\{\begin{matrix}\text{ASORT}\\ \text{DSORT}\end{matrix}\right\}\text{(alpha array name2)}$$

Array 2 is sorted in ascending (ASORT) or descending (DSORT) order into array 1. Array 1 is redimensioned to correspond to array 2 as follows:

Array 2	Array 1	Redimensioned to
(nxm)[L]	(pxq)[k]	(nxm)[L]
(nxm)[L]	(p)[k]	(nmx1)[L]
(n)[L]	(pxq)[k]	(nx1)[L]
(n)[L]	(p)[k]	(nx1)[L]

where n,p= number of rows;
m,q= number of columns.

An error occurs if array 1 is not as large (in bytes) as array 2.

The sorted values are placed in array 1 row-by-row, starting with the first array variable. If array 1 is larger than array 2, remaining locations are unchanged.

As sorting is done directly into array 1, the two arrays may not be the same(i.e., sort-in-place is not supported).

NOTE:

Alphanumeric sorting uses the usual ASCII collating sequence.

Syntax examples:

```
100 MAT A=ASORT(B)
200 MAT A$=DSORT(B$)
300 MAT C$=ASORT(B$)
```

Program example:

```
100 DIM A(3,4),B(2,3),C(7)
200 MAT READ(B)
300 MAT A=ASORT(B)
400 MAT C=DSORT(B)
500 MAT PRINT B,A,C
600 DATA 3,4,7,1,5,2
```

```
Result:  B =    3   4   7
                1   5   2

         A =    1   2   3
                4   5   7

                7
                5
         C =    4
                3
                2
                1
```

MAT CON (MAT CONstant) Statement

```
General Form:

MAT c=CON [(d1[,d2])]

where:
    c is a numeric array name and d1,d2 are expressions
    specifying new dimensions.
    1<d1,d2<32767
```

This statement sets all elements of the specified array to one (1). Using (d1,d2) causes the matrix to be redimensioned. If (d1,d2) are not used, the matrix dimensions are as specified in a previous COM, DIM or MAT statement, or are the default values.

Examples of MAT CON syntax:

```
100 MAT A=CON(10)
200 MAT C=CON(5,7)
300 MAT B=CON(5*Q,S)
400 MAT A=CON
```

Examples showing usage in a program:

```
100 MAT A = CON(2,2)
200 MAT PRINT A;
```

When this program is executed, the CRT displays the result in packed format:

```
1  1
1  1
```

MAT= (MAT assignment) Statement

```
General Form:

MAT a=b

where:
    a and b are both numeric or both
    alphanumeric array names.
```

This statement replaces each element of array a with the corresponding element of array b. Array a is implicitly redimensioned to conform to the dimensions of array b.

Syntax examples:

```
100 DIM A(3,5),B(3,5)
200 MAT A=B
300 DIM C(4,6),D(2,4)
400 MAT C=D
500 DIM E(6),F(7)
600 MAT F=E
```

Example showing use in a program:

```
              1     1     1                    9     8     7
     Let A =  1     1     1       B =          6     5     4
              1     1     1
```

Program:

```
100 DIM A(3,3),B(2,3)
200 MAT A=CON
300 MAT PRINT A
400 MAT INPUT B
500 MAT A=B
600 MAT PRINT A
```

When this program is executed, the constant 3-by-3 array A is displayed as:

```
1     1     1
1     1     1
1     1     1
```

in zoned format; the array B is input via the keyboard, and the new array A is displayed as:

```
9     8     7
6     5     4
```

in zoned format.

MAT IDN (MAT identity) Statement

```
General Form:

MAT c = IDN [(d1,[d2])]

where:
    c is a numeric array name and d1,d2
    are expressions specifying new dimensions.
    1<d1,d2<32767
```

This statement causes the specified matrix to assume the form of the identity matrix. If the specified matrix is not a square matrix, an error occurs and execution is terminated.

Using (d1,d2) causes the matrix to be redimensioned. If (d1,d2) are not used, the matrix has the dimensions specified in a previous COM, DIM or MAT statement.

Syntax example:

```
100 MAT A = IDN(4,4)
200 MAT B = IDN
300 MAT C = IDN(X,Y)
```

Identity Matrix example:

```
100 DIM A(4,4)
200 MAT A = IDN
300 MAT PRINT A
```

When this program is executed, the matrix A is displayed in zoned format as:

```
1     0     0     0
0     1     0     0
0     0     1     0
0     0     0     1
```

MAT INPUT Statement

```
General Form:

MAT INPUT [literal,]
        [ , { numeric array name [(d1[,d2])]                } ] ...
        [   { alpha array name [(d1[,d2])[length]]          } ]

where:
    d = expression specifying a new dimension
    (1<d1,d2<32767).

    length = expression specifying maximum length of each.
             alpha array element (1<length<256).
```

The MAT INPUT statement allows the user to supply values from the keyboard for an array during the running of a program. The MAT INPUT statement displays the literal, if given, and a question mark (?) and waits for the user to supply values for the specified arrays. The dimensions of the array(s) are as last specified in the program (by a COM, DIM or MAT statement), unless the user redimensions the array(s) by specifying the new dimension(s) after the array name(s). The maximum length for alphanumeric array elements can be specified by including the length after the dimensions specification; if no length is specified, a default value of 16 is used.

The values entered are assigned to an array row by row until the array is filled. If more than one value is entered on a line, the values must be separated by commas. Alphanumeric data with leading spaces or commas can be entered by entering a quotation character (") before and after the data value. Several lines can be used to enter the required data. Excess data is ignored. If there is a system-detected error in the entered data, the data must be reentered beginning with the erroneous value. The data which preceded the error is used as previously entered. Input data must be compatible with the array (i.e., numeric data for numeric arrays, alphanumeric literal strings for alphanumeric arrays). Entering no data on an input line (i.e., only keying ENTER to enter a carriage return) causes the remaining elements of the array currently being filled to be ignored.

Example with numeric variables:

```
100 DIM A(2),B(3),C(3,4)
200 MAT INPUT A,B(2),C(2,4)
```

When this program is run, key in the values, separated by commas,

```
-3, -5, .612, .41
```

Press the ENTER key to enter these values for array elements A(1), A(2), B(1) and B(2). Enter the values

```
-6.4, -5.6, 98
```

separated by commas; press ENTER to enter these values for the array elements C(1,1), C(1,2), and C(1,3). Touch the ENTER key without entering further values to enter a carriage return and ignore the rest of the possible values for the array C.

Example with alphanumeric string variables:

```
100 DIM C$(2),A$(4)4,B(3)
200 MAT INPUT A$(4)3,B(2),C$
```

Enter RAD,DEG,MIN,SEC,2.5,5.6,LAST RESULT,"ROTATE X,Y" and Key ENTER.

Result:

```
      RAD
A$ =  DEG       B =  2.5       C$ =  LAST RESULT
      MIN            5.6             ROTATE X,Y
      SEC
```

MAT INV (MAT inverse) Statement

```
General Form:

MAT c = INV(a)[,d]

where:
    c and a = numeric array names.
    d = numeric variable; the value of the
        determinant of the array a.
```

This statement causes the inverse of matrix a to be placed in matrix c. Matrix c is redimensioned to have the same dimensions as matrix a. Matrix a must be a square matrix; matrix c must be a floating-point matrix. If matrix a is singular (i.e., non-invertible) and d is specified, then d will equal zero after MAT INV is encountered. If d is not specified, an error occurs. In either case, c is destroyed. A matrix can be replaced with the inverse of itself.

After inversion, the variable d (if specified) equals the value of the determinant of matrix a.

This statement uses the Gauss-Jordan Elimination Method done in place; as with any matrix inversion technique, results can be inaccurate if the determinant (or normalized determinant) of the matrix is close to zero. It is therefore good practice to check the determinant after any inversion.

The Gauss-Jordan Elimination Method also works best when values on the main diagonal are in the same range as other values in the matrix; in particular, numbers with large negative exponents on the main diagonal should be avoided when other values are not in this range. When in doubt, it is a good plan to check the data before inversion and adjust or rearrange it accordingly (for example, zero elements that are close to zero, or rearrange data so that elements on the main diagonal are as large as possible).

Syntax examples:

```
100 MAT A=INV(B)
200 MAT Z1=INV(P),X2
300 MAT F=INV(C),J3
400 MAT C=INV(C)
```

The following program takes the 4x4 matrix A from the keyboard input, calculates its inverse, and prints both the result and the value of the determinant of A.

```
100 DIM A(4,4)
200 PRINT "ENTER ELEMENTS OF A 4x4 MATRIX"
300 MAT INPUT A
400 MAT B=INV(A),D
500 MAT PRINT B
600 REM B IS THE INVERSE OF A, D IS THE DETERMINANT OF A
700 PRINT "VALUE OF DET.A=";D
```

```
If array A= 0   2   4   8  then array B=   -1    0    0    .25
            0   0   1   0                -3.5   -2   -4      1
            1   0   0   1                   0    1    0      0
            4   8  16  32                   1    0    1   -.25
```

and the value of the determinant of A = -8.

MAT * (MAT multiplication) Statement

```
General Form:

MAT c=a*b

where:
    c, a, and b are numeric array names.
```

The product of arrays a and b is stored in array c. Array c cannot appear on both sides of the equation but a and b may be identical. If the number of columns in matrix a does not equal the number of rows in matrix b, an error occurs and execution is terminated. The resulting dimension of c is determined by the number of rows in a and the number of columns in b.

Syntax example:

```
100 DIM A(5,2),B(2,3),C(4,7)
200 DIM E(3,4),F(4,7),G(3,7)
300 MAT G = E * F
400 MAT C = A * B
```

Program Usage example:

```
100 DIM A(2,3),B(3,4)
200 MAT INPUT A,B
300 MAT C = A * B
400 MAT PRINT C
```

```
Let A =   0   1   4   , B =   5   1   0   4
          7   7   7           4   1   0   4
                              3   4   3   4
```

When the program is executed and arrays A and B are entered, array C is displayed as:

```
16     17     12     20
84     42     21     84
```

MAT PRINT Statement

```
General Form:

MAT PRINT array name [t array name] ... [t]

where:
    t is a comma or semicolon.
```

The MAT PRINT statement prints arrays in the order given in the statement. Each matrix is printed row by row. All elements of a row are printed on as many lines as required. A multiple MAT PRINT is treated like several single MAT PRINTs. Numeric arrays are printed in zoned format unless the array name is followed by a semicolon, in which case the array is printed in packed format. For alphanumeric arrays, the zone length is set equal to the maximum length defined for each array element (not always 16). A vector (a one-dimensional array) is printed as a column vector.

Syntax example:

```
100 DIM A(4),B(2,4),B$(10),C$(6)
200 MAT PRINT A;B,C$
300 MAT PRINT A,B$
```

Program Usage example:

This program takes nine alphanumeric quantities as input, each up to 16 characters long, and prints them as a 3x3 array in packed format.

```
100 DIM Z$(3,3)
200 MAT INPUT Z$
300 MAT PRINT Z$;
```

At the workstation enter the values:

```
A, B, C, D, E, F, G, H, I
```

Results:

```
ABC
DEF
GHI
```

MAT READ Statement

```
General Form:

MAT READ [{numeric array name}[(d1[,d2])]          ],...
         [{alpha array name  }[(d1[,d2])[length]]]

where:
    d = expression specifying a new dimension
        (1<d1,d2<32767).

    length = expression specifying maximum length of each
             alpha array element
             (1<length<256).
```

The MAT READ statement is used to assign values contained in DATA statements to array variables without referencing each member of the array individually. The MAT READ statement causes the referenced arrays to be filled sequentially with the values available from the DATA statement(s). Each array is filled row by row. Values are retrieved from a DATA statement in the order they occur on that program line. If a MAT READ statement references beyond the limit of existing values in a DATA statement, the next sequential DATA statement is used. If no more DATA statements are in the program, an error occurs and execution is terminated.

Alphanumeric string arrays can also be used in the list. The information entered in the data statement must be compatible with the array (i.e., numeric values for numeric arrays, alphanumeric literals for alphanumeric arrays).

The dimensions of the array(s) are as last specified in the program (by a COM, DIM, or MAT statement), unless the user redimensions the array(s) by specifying new dimension(s) after the array name(s) in the MAT READ statement. The maximum length for alphanumeric array elements can be specified by including the length after the dimension specification; when no length is specified, a default of 16 is used.

Program example:

```
100 DIM A(1),B(3,3)
200 MAT READ A,B(2,3)
300 DATA 1, -.2,315, -.398, 6.21, 0, 0
400 MAT PRINT A,B
```

Result:

```
A =   1          B =   -.2         315      -.398
                       6.21        0        0
```

MAT REDIM Statement

```
General Form:

MAT REDIM redim-elt[,redim-elt]...

where:
    redim-elt ={numeric array name}(exp1[,exp2])
               {alpha array name  }(exp1[,exp2])[exp3]

    1 < exp1 < 32767
    1 < exp2 < 32767
    1 < exp3 < 256
```

The MAT REDIM statement redimensions the specified arrays to the dimensions specified by the expressions. The rules are like DIM, except as follows:

1. As indicated, alpha scalars may not be REDIMed.

2. MAT REDIM may occur anywhere in the program or subprogram. Its only effect is to change the dimensions and lengths of the specified array; it does not affect the values currently assigned to array elements.

3. The total (byte) space required for the array must be no greater than that initially allotted to it by DIM or default (10x10, len=16 for alpha arrays).

4. If exp3 is omitted, it is set to 16, regardless of the previous length.

5. A matrix may not be redimensioned as a vector, or vice versa.

Syntax examples:

```
100 MAT REDIM A(10),B$(10,20)10
200 MAT REDIM A(20,30)
```

MAT()* (MAT scalar multiplication) Statement

```
General Form:

MAT c = (k) * a

where:
    c and a are numeric array names and k is an expression.
```

Each element of the array a is multiplied by the value of expression k and the product is stored in array c. Array c can appear on both sides of the equation. Array c is redimensioned to the same dimensions as array a.

Syntax examples:

```
100 MAT C = (SIN(X))*A
200 MAT D = (X+Y*2)*A
300 MAT A = (5)*A
```

Program example:

This program inputs a 3 by 3 array and a scalar. It then performs scalar multiplication and displays the result.

```
100 PRINT "ENTER DATA FOR A 3x3 ARRAY"
200 MAT INPUT C(3,3)
300 PRINT "ENTER SCALAR"
400 INPUT K
500 MAT A = (K)*C
600 MAT PRINT A;
```

```
Let C =  5   3   1    , K = 5    then A =  25   15    5
         2   2   2                         10   10   10
         1   1   1                          5    5    5
```

MAT - (MAT subtraction) Statement

General Form:

MAT c = a - b

where:
    a, b, and c are numeric array names.

This statement subtracts numeric arrays of the same dimension. The difference of each pair of elements is stored in the corresponding element of c. Any two or all of a, b, and c may be the same. An error occurs and execution is terminated if the dimensions of a and b are not the same. Array c is redimensioned to have the same dimensions as arrays a and b.

Syntax example:

```
100 DIM A(6,3),B(6,3),C(6,3),D(4),E(4)
200 MAT C = A - B
300 MAT C = A - C
400 MAT D = D - E
```

Program example

```
100 DIM D(3,3), E(3,3)
200 MAT INPUT D
300 MAT INPUT E
400 MAT F = D - E
500 MAT PRINT F
```

```
If D= 1  1  1   , E= 3  3  3, then F= -2  -2  -2
      1  1  1        3  3  3          -2  -2  -2
      2  2  2        3  3  3          -1  -1  -1
```

MAT TRN (transpose) Statement

```
General Form:

MAT c = TRN(a)

where:
    a and c are array names   (both numeric or both alphanumeric).
```

This statement causes array c to be replaced by the transpose of array a. Array c is redimensioned to the same dimensions as the transpose of array a. Array c cannot appear on both sides of the equation.

Syntax example:

```
100 MAT C = TRN(A)
```

Program Usage example:

```
100 DIM A(3,3)
200 MAT INPUT A
300 MAT C = TRN(A)
400 MAT PRINT C
```

```
Let A =    9   8   7
           6   5   4
           3   2   1
```

When the program is executed, C is displayed as:

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

MAT ZER (MAT ZERO) Statement

```
General Form:

MAT c = ZER [(dl[,d2])]

where:
    c is a numeric array name and dl,d2 are expressions
    specifying new dimensions.  (1 < dl,d2 < 32767)
```

This statement sets all elements of the specified array equal to zero. Using (dl, d2) causes the matrix to be redimensioned. If (dl,d2) are not used, the matrix retains the dimensions specified in a previous COM, DIM, or MAT statement.

Syntax example:

```
100 MAT C = ZER(5,2)
200 MAT B = ZER
300 MAT A = ZER(F,T+2)
400 MAT D = ZER(20)
```

MAX Function

```
General Form:

MAX  (exp[,exp]    )

where:
    exp = a numeric scalar or numeric array.
```

The MAX function returns the largest element in the argument list, or in the case of an array, the largest element of the array.

Syntax examples:

```
     100 A=MAX(B,C,D)
     200 D=MAX(E())
```

Mathematical Functions

The following General Form 1 applies to most mathematical functions. General Forms 2-4 are listed after the discussion of General Form 1.

```
General Form 1:

function (exp)

where:

    function =    SIN
                  COS
                  TAN
                  ARCSIN
                  ARCCOS
                  ARCTAN
                  ATN
                  ABS
                  EXP
                  INT
                  LGT
                  LOG
                  SGN
                  SQR
```

Trigonometric Functions

The sine, cosine, tangent, arcsine, arccosine, and arctangent functions are available in BASIC. Other trigonometric functions can be easily expressed using these functions in expressions. (When using these functions in combination, care must be taken to avoid significant data conversion errors. See Appendix C, Numeric Data Representation in VS BASIC, for a complete discussion.)

Function Name	Sample Expression	Meaning
SIN	SIN(X)	the sine of the argument
COS	COS(X)	the cosine of the argument
TAN	TAN(X)	the tangent of the argument
ARCSIN	ARCSIN(X)	the inverse sine of the argument
ARCCOS	ARCCOS(X)	the inverse cosine of the argument
ARCTAN	ARCTAN(X)	the inverse tangent of the argument
ATN	ATN(X)	synonym for ARCTAN.

Other Numerical Functions

The remaining numerical functions are described below.

Function Name	Sample Expression	Meaning
ABS	ABS(X)	The absolute value of the argument: -X if X < 0; X if X > = 0.
SQR	SQR(X)	The square root of the argument; X raised to the .5 power.
EXP	EXP(X)	The exponential function; "e" (2.718...) raised to the X-th power.
INT	INT(X)	The greatest-integer function; the greatest integer less than or equal to the argument.
LGT	LGT(X)	Common (base 10) logarithm.
LOG	LOG(X)	Natural (base "e") logarithm; inverse function of EXP.
SGN	SGN(X)	The signum function; -1 if the argument is negative; 0 if the argument is zero; +1 if the argument is positive.

```
General Form 2:

function (exp[,exp]...)

where:

    function =    MAX
                  MIN
```

Function Name	Sample Expression	Meaning
MAX	MAX(X,Y,Z)	The value of the largest element in the argument list.
MIN	MIN(X,Y,Z)	The value of the smallest element in the argument list.

```
General Form 3:

MOD(exp,exp)
```

Function Name	Sample Expression	Meaning
MOD	MOD(X,Y)	The modulus function; the remainder of the division of the first element by the second.

```
General Form 4:

PI
```

Function Name	Sample Expression	Meaning
PI	PI	The value 3.14159265358979323.

See Section 2.6 for more information on Numeric Functions.

MIN Function

General Form:

```
MIN  (exp[,exp]    )
```

where:
    exp = a numeric scalar or numeric array.

The MIN function returns the smallest element in the argument list or array.

Syntax examples:

```
100 MIN (B,C,D)
200 MIN (E())
```

MOD Function

General Form:

MOD(numeric expression, numeric expression)

The MOD (modulus) function returns a numeric value equal to the remainder of the division of the first expression by the second. The value returned is an integer value if both expressions are integers; otherwise it is floating-point.

Example:

```
MOD(5,3) = 2
```

NEXT Statement

```
General Form:

NEXT numeric scalar variable [,numeric scalar variable]...
```

The NEXT statement defines the end of a FOR...NEXT loop; it must contain the same index variable(s) as a previously executed FOR statement. A multiple NEXT is executed left to right, i.e.,

```
NEXT I,J,K
```

is equivalent to

```
NEXT I
NEXT J
NEXT K
```

When a FOR...NEXT loop is encountered, the index variable takes the value initially assigned in the FOR statement. When the NEXT statement is executed, the STEP value specified in the FOR statement is added to the value of the index. (If no STEP value is given, +1 is used.) If the result is within the range specified in the FOR statement, the result (index + STEP) is assigned to the index variable and execution continues at the statement following the FOR statement. If the result is outside the range specified in the FOR statement, the index variable is unaltered and execution passes to the statement following the NEXT statement. The FOR...NEXT loop is then considered completed. A subsequent NEXT with the same index variable which is encountered without first encountering a FOR with the same index variable will produce a runtime error.

NUM Function

General Form:

NUM (alpha-expression)

The NUM function determines the number of sequential ASCII characters in the specified alpha-expression that represents a legal BASIC number. A numeric character is defined to be one of the following: digits 0 through 9, and special characters E, . (decimal point), +, -, space (provided the space is non-embedded; leading and trailing spaces are considered numeric characters, embedded spaces are not). (% is not a legal numeric character.) Numeric characters are counted starting with the first character of the alpha expression. The count is ended either by the occurrence of a non-numeric character, or when the sequence of numeric characters fails to conform to standard BASIC number format. Leading and trailing spaces are included in the count. Thus, NUM can be used to verify that an alphanumeric value is a legitimate BASIC representation of a numeric value, or to determine the length of a numeric portion of an alphanumeric value. NUM can be used wherever numeric functions are normally used. NUM is particularly useful in applications where it is desirable to numerically validate input data under program control. If A$ = "1E88", then NUM(A$)=16 even though 1E88 is an illegal value, since 1E88 exceeds the legal size for a floating point constant. This occurs because NUM checks only format, not value.

The result of the NUM function is an integer.

Examples:

```
100 A$ = "98.7+53.6"
200 X=NUM(A$)
```

Note: X=4 since the sequence of numeric characters fails to conform to standard BASIC number format when the + character is encountered.

```
100 INPUT A$
200 IF NUM(A$)=16 THEN 500
300 PRINT"NON-NUMERIC,ENTER AGAIN"
400 GOTO 100
500 CONVERT A$ TO X
600 PRINT "X=";X
Run program:
? 123A5
NON-NUMERIC, ENTER AGAIN
? 12345
X=12345
```

Note: The program illustrates how numeric information can be entered as a character string, numerically validated, and then converted to an internal number.

ON Statement

General Form:

$$\text{ON expression} \begin{Bmatrix}\text{GOTO}\\ \text{GOSUB}\end{Bmatrix} \text{entry [,entry]...}$$

where:

$$\text{entry} = \begin{Bmatrix}\text{line number}\\ \text{null}\\ \text{statement label}\end{Bmatrix}$$

The last entry must be a line number or a statement label(no trailing commas).

The ON statement is a computed GOTO or GOSUB statement.

If I is the truncated value of the expression, transfer is determined by the Ith entry:

1. If a line number or statement label, the transfer is made to that line or statement.
2. If null, no transfer is made.
3. If I<1 or > number of entries, no transfer is made.

In Options 2 or 3 above, execution continues at the next executable statement, e.g., ON X GOTO,,100,200,,300,,,400

Value of X	Transfer
-2	none
-1	none
0	none
1	none
2	none
3	100
4	200
5	none
6	300
7	none
8	none
9	400
10	none
11	none

OPEN Statement

General Form:

```
                                                  ⎧INPUT  ⎫
                                                  ⎪IO     ⎪
OPEN [,] ⎡⎧NODISPLAY ⎫[,]⎤ file-exp[,]   ⎨SHARED ⎬
         ⎣⎩NOGETPARM ⎭   ⎦                ⎪EXTEND ⎪
                                                  ⎩OUTPUT ⎭

[,SPACE = num-exp1] [,DPACK = num-exp2] [,IPACK = num-exp3]

[,FILE = alpha-exp1] [,LIBRARY = alpha-exp2]

[,VOLUME = alpha-exp3] [,FILESEQ = num-exp4]

[,BLOCKS = num-exp5]
```

where:

alpha-exp1,2,3	= file, library, and volume names must be enclosed in quotation marks.
Filename	= at most 8 characters (remainder ignored).
Library	= at most 8 characters (remainder ignored).
Volume	= at most 6 characters (remainder ignored).
num-exp5	= size of I/O buffer (in blocks of 2048 bytes). default = 1 block

(use of other parameters explained below).

OPEN is used to open an existing disk or tape file or create a new file. (The OPEN statement is discussed in detail in Subsection 8.3.2, The OPEN and CLOSE Statements.) The file number (provided by file-exp) must have appeared in a SELECT statement (see SELECT). BLOCKS is optional, but file, library, and volume names will be requested by the system (using the SELECT prname) even if included in OPEN, unless the file was OPENed and CLOSEd previously or NOGETPARM or NODISPLAY was specified.

The various OPEN modes for old and new files, and the allowed I/O operations are listed in Table P2-5.

Attempting to OPEN a file that has already been OPENed and not yet CLOSEd causes an irrecoverable error at run-time.

Use of the SPACE, DPACK, IPACK, NODISPLAY, and NOGETPARM fields is explained below.

NODISPLAY, NOGETPARM

When OPENing a file in the program, OPEN will normally issue a GETPARM (see the discussion of GETPARM in the Programmer's Introduction) to the workstation or procedure, requesting the FILE, LIBRARY, and VOLUME parameters.

The prompt at the workstation can be suppressed by specifying NODISPLAY. This should only be done if the correct FILE, LIBRARY, and VOLUME have been specified in this or a previous OPEN, or in a procedure, or if SET defaults are in use. (For a discussion of SET usage constants, see the Programmer's Introduction.)

Both the workstation prompt and the procedure file prompt may be suppressed by specifying NOGETPARM. This should not be done if the file parameters are to be accessible/modifiable from a user procedure. (For a discussion of procedures, see Chapter 7 of the VS Programmer's Introduction.)

The remaining parameters differ in usage depending on whether the file is being OPENed in OUTPUT or non-OUTPUT mode.

SPACE

OUTPUT:	Specifies the approximate number of records to be put in the new file. If OUTPUT is not specified, a GETPARM will be displayed.
non-OUTPUT:	If a variable (i.e., a receiver), it will contain the number of records currently in the file after OPEN.

DPACK, IPACK

OUTPUT:	Specifies the block packing densities (integer) for the records/keys, respectively, for a new INDEXED file only.
non-OUTPUT:	Ignored

In any mode, if FILE/LIBRARY/VOLUME are alpha-receivers, the actual names will be returned to the receivers after the OPEN statement is completed.

FILESEQ

File Sequence number. For tape files only.

Table P2-5.

Legal Function Requests and Descriptions

TYPE MODE	CONSEC	VAR CONSEC	INDEXED VAR INDEXED	TAPE	PRINTER
INPUT (old files only)	Ops: READ,SKIP Consecutive or relative READ or SKIP, starting from beginning of the file.	Ops: READ,SKIP Consecutive or relative READ or SKIP starting from beginning of file.	Ops: READ Consecutive or keyed READ, starting from beginning or after last record read.	Ops: READ,SKIP Consecutive or relative READ or SKIP starting. from beginning of file.	NOT ALLOWED
IO (old files only)	Ops: READ,REWRITE, SKIP Consecutive or relative READ or SKIP from beginning of file, with HOLD/ REWRITE option.	Ops: READ,SKIP, REWRITE Consecutive or relative READ or SKIP from beginning of file, with HOLD/REWRITE option.	Ops: READ,WRITE, REWRITE,DELETE Consecutive or keyed READ or WRITE from beginning of (key) file, with HOLD/ REWRITE/DELETE option.	Ops: READ,SKIP Consecutive or relative READ or SKIP from the beginning of the file with HOLD option.	NOT ALLOWED
SHARED (old INDEXED files; old or new CONSEC files)	NOT ALLOWED	Ops: WRITE, HOLD, RELEASE Used for (variable length) <u>log files</u>.	Ops: READ,WRITE, REWRITE,DELETE HOLD, RELEASE Same as IO, but allows multiple access, independently, HOLD protection.	NOT ALLOWED	NOT ALLOWED
OUTPUT (new files only) (old files deleted)	Ops: WRITE Writes records consecutively to a new file.	Ops: WRITE Writes records consecutively to a new file.	Ops: WRITE Writes records to a new file - (primary) keys must be in <u>ascending</u> order.	Ops: WRITE Writes records consecutively to a new file.	Ops: WRITE Writes records consecutively to a new file.
EXTEND (old files only)	Ops: WRITE Writes records consecutively, starting at the current file end.	Ops: WRITE Writes records consecutively, starting at the current file end.	NOT ALLOWED	NOT ALLOWED	NOT ALLOWED

Syntax examples:

```
100 OPEN NODISPLAY #1,IO,FILE="THOMAS",LIBRARY="STEARNS",!
200 VOLUME="ELIOT"
300 OPEN #2,OUTPUT
400 OPEN NODISPLAY #3, INPUT, VOLUME="TAPE1", FILESEQ=1
```

OR Logical Operator

General Form:

[LET] alpha-receiver = [logical exp] OR logical exp

logical exp -- see Section 5.7, Logical Expressions.

The OR operator logically OR's two or more alphanumeric arguments.

Example:

```
100 A$= "SAINT"
200 B$= "S    "
300 C$= A$ OR B$
400 PRINT C$
```

```
Output:  Saint
```

Capital A is HEX(41) or 01000001 in binary, a blank space is HEX(20) or 00100000 in binary. When two characters are ORed, a binary one in either becomes a binary one in the result. Thus, ORing A with " " produces binary 01100001 or HEX(61), which is the ASCII "a".

The operation proceeds from left to right. If the operand (logical expression) is shorter than the receiver, the remaining characters of the receiver are unchanged. If the operand is longer than the receiver, the operation stops when the receiver is exhausted.

See Section 5.7, Logical Expressions, for more information on logical expressions.

PACK

```
General Form:

PACK PIC (image) alpha-receiver FROM

{numeric array-designator} [,{numeric array-designator  ...}]
{    expression           }  [ {      expression            }]

where:
    image = [±] [#...][.][#...][↑↑↑↑] (at least 1 "#")
```

The PACK statement packs numeric values into an alphanumeric receiver, reducing the storage requirements for large amounts of numeric data where only a few significant digits are required. The specified numeric values are formatted into packed decimal form (two digits per byte) according to the format specified by the image, and stored sequentially into the specified alphanumeric receiver. Receivers are filled from the first byte until all numeric data has been stored. An entire numeric array can be packed by specifying the array with a numeric array-designator (e.g., N()). An error will result if the receiver is not large enough to store all the numeric values to be packed.

The image is composed of # characters to signify digits and, optionally, +, -, ., and   characters to specify sign, decimal point position, and exponential format. The image can be classified into two general formats.

Format	Example
Fixed Point	##.##
Exponential	#.##

Numeric values are packed according to the following rules:

1. Two digits are packed per byte. A digit is stored for each # in the image.

2. If a sign (+ or -) is specified, it occupies the high-order 1/2 byte. A single hex digit is used to represent both the sign of the number and the sign of the exponent for exponential images. The four bits of this hex digit are set as follows:

   Bit 1 (leftmost) set to 1 if exponent is negative.
   Bit 2 OFF ("0").
   Bit 3 OFF ("0").
   Bit 4 (rightmost) set to 1 if number is negative.

3. If no sign is specified, the absolute value of the number is stored, and the sign of the exponent is assumed to be plus (+).

4. The decimal point is not stored. When unpacking the data (see UNPACK), the decimal point position is specified in the image.

5. The packed numeric value is left-justified in the alpha-receiver, with the sign digit (if specified) occupying the high-order half-byte, followed by the number in packed decimal format (two digits per byte). The exponent occupies the two low-order half-bytes (if specified). The packed value always requires a whole number of bytes, even if the image calls for other than a whole number. For example, the image '###' calls for 1 1/2 bytes, but 2 bytes are required. In such cases, the value of the unused half-byte (the low-order half-byte) is not altered by the PACK operation.

6. If the image has format 1, the value is edited as a fixed point number, truncating or extending with zeros any fraction and inserting leading zeros for nonsignificant integer digits according to the image specification. An error results if the number of integer digits exceeds the format specification.

7. If the image has format 2, the value is edited as an exponential number. The value is scaled as specified by the image (there are no leading zeros). The exponent occupies one byte, and is stored as the two low-order hex digits in the packed value.

Example of storage requirments:

####	= 2 bytes
###	= 2 bytes
+##.###	= 3 bytes
+#.##	= 2 bytes

Syntax examples:

```
100 PACK PIC (####)A$ FROM X
200 PACK PIC (####)STR(A$,4,2) FROM N(1)
300 PACK PIC (##.##)A1$() FROM X,Y,N(),M()
```

```
General Forms:

$PACK [( [{D=} ] alpha-exp)]  alpha-receiver FROM arg[,arg]...
         {F=}

                 [,DATA {GOTO } {line number     }]
                        {GOSUB} {statement label }
where:
   {line number    } = line number or statement label of
   {statement label}   data conversion error exit.

     arg = {expression                                  }
           {alpha-expression, EXCEPT alpha array string }
           {array-designator                            }

$UNPACK [( [{D=} ] alpha-exp)]alpha-expression TO arg[,arg]...
            {F=}

                [,DATA {GOTO } {line number     }]
                       {GOSUB} {statement label }
where:
     arg ={receiver, EXCEPT alpha array string}
          {array-designator                   }
```

$PACK and $UNPACK pack and unpack numeric and character data, in any of several formats specified by the user, into the alpha receiver or from the alpha-expression, respectively.

Concerning the operation of $PACK and $UNPACK in general:

1. An arg of the form "name$()" is always recognized as an array of elements, never as an alpha array string. Use a STR to get an array string.

2. Array elements are generally considered as individual consecutive values or receivers (row-by-row). The exception is F format, where a single format applies to all of the array elements.

3. $PACK generates an error (or exit) if the alpha receiver is not long enough to store all of the args in the specified format. This is true with any of the formats.

4. $UNPACK generates an error (or exit) if the unpacked data is not the same type (alpha, numeric) as the receiver.

Delimiter Format

Indicated by the presence of "D = alpha-expression". The format of the $PACKed data is:

data	DEL	data	DEL	...	data	DEL

where DEL is the user-specified delimiter.

The alpha-expression following "D=" must contain at least 2 bytes:

byte 1 =	Conversion code. This is used only by $UNPACK, but must have one of the four legal values for either $PACK or $UNPACK:
byte 2 =	(DEL) delimiter character.

Hex Value	(UNPACK) Result
00	A) Error if insufficient data in the buffer. B) Skip a receiver (or array element) for each extra delimiter encountered.
01	A) No error if insufficient data in the buffer -- remaining receivers are left unchanged. B) Skip a receiver for each extra delimiter.
02	A) Error if insufficient buffer data. B) Ignore extra delimiters.
03	A) No error if insufficient buffer data. B) Ignore extra delimiters.

1. $PACK:

   The general form is as diagrammed above. The structure of the data entries is as follows:

   Numeric -- Exactly like PRINT, without the trailing blank.

   Alphanumeric -- Defined length is stored.

2. $UNPACK:

Extra delimiters may be present, as described in the conversion code above. A missing final delimiter causes the last data value to be considered as extending to the (defined) end of the buffer (alpha-expression). Specific data entries allowed:

Numeric -- Allows any numeric constant which would be allowed on a program line, including leading and trailing blanks. Exception: As with CONVERT, "%" is not recognized as a legal character.

Alphanumeric -- Anything, any length. It will be right-padded or truncated to fit the receiver.

NOTES:

$UNPACK condition code may be set not to cause an error if there are not enough data values in the buffer; this is true only of delimiter (D) format. Any of the other formats will cause an error (or DATA exit) if the buffer has insufficient data.

Any errors incurred in executing $PACK and $UNPACK do not affect values already packed or unpacked in the same statement, i.e., the error occurs only when the first erroneous conversion is encountered. This is like the regular PACK and UNPACK statements.

Field Format

Indicated by the presence of "F = alpha-expression". The format of the $PACKed data is:

field	field	field	...	field	field

where field = a skip field
a formatted data value

The alpha-expression following "F=" must contain at least as many pairs of bytes as there are args in the arg list. (Each pair corresponds to an arg, whether it is a scalar or array arg.)

From left to right, each arg has a corresponding byte pair, in which:

```
byte 2 = field width (bytes) in hex >0
byte 1 = field type
            {00  (skip field)
            {10  (free-format)
            {2h  (ASCII integer format)
            {p
            {3h  (IBM display format)
            {p
            {4h  (WANG display format)
            {p
            {5h  (IBM packed decimal format)
            {p
            {A0  (alphanumeric field)
```

Field Types

1. 00 Skip

   In either $PACK or $UNPACK, skips the specified number of bytes in the buffer; skipped characters are unchanged.

2. A0 Alphanumeric

   For alphanumeric data; in either $PACK or $UNPACK, the value is padded or truncated on the right to fit the field or receiver, respectively.

3. 10 Free-format ASCII numeric

   $PACK: Same as delimiter format, i.e., same as PRINT, but right-padded or truncated to fit the field.

   $UNPACK: Same as delimiter $UNPACK fields.

4. 2h ASCII Implied decimal
   p

   Form:

s	d	d	d	...	d	d

   d = (ASCII) digit 0-9
   s = sign byte

   $PACK: format as shown; sign byte is ASCII ("+" = HEX(2B), "-" = HEX(2D)).

$UNPACK: format as shown; all the zone half-bytes are ignored and thus may have any value.

5. $\underline{3h_{\underline{p}}}$ <u>IBM numeric display format</u>

Form:

Fh	Fh	Fh	...	Fh	Fh	h h_S

h = hexdigit 0-9 only
h_s = sign digit

$PACK: format as shown

h_s = C (+)
D (-)

$UNPACK: format as shown; Fs are ignored. h_s may be A,C,E,F (+)
B,D (-)

6. $\underline{4h_{\underline{p}}}$ <u>WANG VS display format</u>

Form:

3h	3h	3h	...	3h	3h	h h_s

h = hexdigit 0-9 only
h_s = sign digit

$PACK: format as shown

h_s = F (+)
D (-)

$UNPACK: format as shown; 3s ignored

h_s = D (-)
all else (+)

7. $\underline{5h_{\underline{p}}}$ <u>IBM packed decimal format</u>

Form:

hh	hh	hh	...	hh	hh	hh_s

h = hexdigit 0-9 only
h_s = sign digit

$PACK: format as shown; h_s = C (+)
D (-)

$UNPACK: format as shown; h_s = A,C,E,F (+)
B,D (-)

Field types 2-5 have the following characteristics in common:

In $PACK, an overflow causes an error, but the field is filled with zeros and the correct sign.

In $2h_p$, $3h_p$, and $4h_p$ zoned format, zones are not checked when $UNPACKed, and thus may take on any values. This includes the zone of the sign byte in 2h format. One consequence of this is that blanks are interpreted as zoned zeroes in $2h_p$, $3h_p$, $4h_p$.

h in the format specification denotes the number of p digits to the right of the implied decimal point. It may take on any hexdigit value, and may be larger than the number of digits in the field (in which case leading decimal zeroes are implied).

In $PACK, an underflow causes no error and fills the field with zeros and a + sign, regardless of the sign of the expression itself.

$PACK inserts leading and trailing zeros where necessary.

$UNPACK allows any number of digits; only the first 15 significant digits are used; the rest only serve to position the decimal point.

2200 Disk Storage Format

Indicated by the absence of both D and F.

Form:

		SOV	data	SOV	data	...	SOV	data	EOB

control

where:

CONTROL = Pseudo-2200 control bytes (2)
SOV = 2200 Start-of-value byte for the next data value

```
    =  ___________________________________________
       |////|    |    |    |    |    |    |    |
       -------------------------------------------
         S <---------------L------------------->
```

where:
    S = 0 = numeric   L = length in bytes (binary)
        1 = alpha

   data = numeric or alpha value
    EOB = end-of-block byte
        = HEX(FD)

$PACK Data Format

1. Numeric

   Form:

$h_s h_u$	$h_t h_1$	$h_2 h_3$	$h_4 h_5$	$h_6 h_7$	$h_8 h_9$	$h_{10} h_{11}$	$h_{12} h_{13}$

   Value is decimal floating point.

   h_s = sign indicator
      = 0, number +, exponent +
        1, number -, exponent +
        8, number +, exponent -
        9, number -, exponent -

   $h_u h_t$ = exponent (units before tens)

   h_1 to h_{13} = mantissa, in the usual order, with the decimal point assumed between h_1 and h_2.

2. Alphanumeric

   Form:

```
   | c | c | c | ... | c | c |
```

   The defined length is stored.

3. Control bytes -- HEX(8001)

$UNPACK Data Format

1. Numeric -- Same as $PACK, but allows any sign digit:

$PACK	$UNPACK
0	0,2,4,6
1	1,3,5,7
8	8,A,C,E
9	9,B,D,F

This occurs because the 2 middle bits of the hexdigit are ignored.

2. Alphanumeric -- Any length, padded or truncated on the right to fit the receiver.

3. Control bytes -- Ignored.

PI Intrinsic Constant

```
General Form:

PI
```

The intrinsic constant PI may appear anywhere a numeric expression may appear. It has the value 3.14159265358979323.

POS Function

General Form:

$$\text{POS ([-] alpha-expression} \left\{\begin{matrix} < \\ <= \\ > \\ >= \\ <> \\ = \end{matrix}\right\} \text{alpha-expression)}$$

The POS function searches the first alpha-expression for a character that is <, <=, >, >=, <>, or = the first character of the second alpha-expression and outputs the location (leftmost=1) of the first such character found. The basis of comparison is the ASCII codes of the characters. POS searches the entire defined length of the alpha-expression.

If no '-' is present, the search executes from left to right, thus outputting the position of the leftmost such character. If "-" is present, the search executes from right to left, thus outputting the position of the rightmost character.

If no character satisfies the condition, POS=0. The output of POS is an integer.

Syntax examples:

```
100 A%=POS(-A$<STR(B$,2,2))
200 FOR A=1 TO 10 STEP POS(C$=B$)
```

PRINT Statement

```
General Form:
       ┌ ┌ ┌USING ┌line number┐┐ [, expression] ┌┌┌,┐ [expression]┐... ┌┌,┐┐ ┐ ┐
       │ │ │      └stmt label ┘┘                 └└└;┘              ┘    └└;┘┘ │ │
PRINT  │<                                                                      > │
       │ │ [prt-elt] ┌┌,┐ [prt-elt]┐ ... ┌┌,┐┐                                 │ │
       └ └           └└;┘          ┘     └└;┘┘                                 ┘ ┘

where:
    prt-elt =  ┌character prt-elt┐
               └control prt -elt ┘

    character prt-elt =  ┌numeric expression       ┐
                        <  alpha expression          >
                         └HEXOF(alpha expression)   ┘

    control prt-elt =  ┌BELL                                    ┐
                       │PAGE                                    │
                      < SKIP[(num-exp)]                          >
                       │TAB(num-exp)                            │
                       │COL(num-exp)                            │
                       └AT(num-exp, num-exp [,    num-exp    ])┘
```

The PRINT statement routes output to either the workstation or printer, depending on which device is currently selected (see SELECT statement).

The placement and format of the data which are output can be controlled either by the use of auxiliary format control (FMT and Image (%)) statements, or by the use of the control print elements described below. If a format control statement is used, the PRINT statement must contain a USING clause referring to the FMT or Image (%) by line number or statement label.

In either case, output begins at the current print position, as determined by the last output operation to the selected device (current print position is indicated on the workstation by the position of the cursor). After the PRINT statement has been executed, the new current print position depends on how the PRINT statement ends. If the PRINT statement ends in a semicolon, a comma, or a control print-element, the current print position is the first position after the last character output; otherwise, the current print position is the first position of the next line.

For PRINT output, the output line is divided into as many zones of 18 characters as possible; thus a 132-column printer has seven zones and the workstation has four. Note that the last zone may be longer than the rest, extending to the end of the line.

Expressions and other print-elements in a PRINT statement must be separated by either commas or semicolons. If USING is specified, these are equivalent and serve only to delimit one expression from the next. If USING is omitted, however, a comma after a character print-element causes the next print-element to begin at the start of the next zone (if the current print position is already in the last zone of a line, the next print-element starts at the beginning of the next line). A semicolon causes no change in print position. After a control print-element, commas and semicolons are equivalent.

A line is not sent to the printer until either the print position is moved beyond the line or until a SKIP(0) is encountered. See Chapter 7, Workstation and Printer Input/Output, for more information on output to the workstation and printer.

Note that the following discussion of print-elements does not apply to PRINT statements which specify USING. For details on the operation of the USING clause, see the entries for the FMT and Image (%) statements.

Print Elements

1. Numeric Expression

   a. If $|\text{exp}|<10^{-1}$ or $|\text{exp}|\geq 10^{15}$, the format is exponential:

   ```
   SMDMMMMMMMMMME ± XXb
   ```

   where:
   ```
    S = minus sign if negative, blank otherwise.
    M = mantissa digit.
    D = decimal point.
   XX = exponent digits.
    b = blank.
   ```

   ```
   e.g., PRINT .000074679;
          b7.4679000000E-05b
         start           end
   ```

   b. If $10^{-1} \leq |\text{exp}| < 10^{15}$, the format is fixed-point:

   ```
   S[Z...][DF...]b
   ```

   where:
   ```
   S = minus sign if negative, blank otherwise.
   Z = zoned digit.
   D = decimal point.
   F = fixed digit.
   b = trailing blank.
   ```

   and total Zs + total Fs $\leq$ 15.

Leading zeros and trailing (decimal) zeros are not printed (but zero is printed as 'b0b'). Up to 15 digits plus a decimal point, if any, are printed.

2. Alpha-expression

The actual length of the alpha-expression is printed. "Actual" implies that trailing blanks of an alpha variable or array string are not printed.

3. HEXOF (alpha-expression)

The hex value (defined length) of the alpha-expression is printed. (This includes trailing blanks -- HEX(20).)

```
    e.g.,   100 DIM A$ 5
            200 A$ = "ABC"
            300 PRINT "HEX VALUE OF A$ =";HEXOF (A$)
Result: HEX VALUE OF A$ = 4142432020
```

4. PAGE

Printer: Advances to line 1, column 1 of a new page.
Workstation: Clears the screen and homes the cursor.

5. BELL

Printer: Ignored.
Workstation: Sounds the workstation alarm; screen and cursor unaffected.

6. SKIP [(n)]

Printer: Advances the print position to column 1 of the nth line after the current line (default n=1).

If n=0, the current line is printed with a carriage return but no linefeed, thus causing the next line to overprint.

If n<0, SKIP is ignored.

Workstation: Advances the cursor print position to column 1 of the nth line after the current line (default=1).

If n>0, the cursor moves down n lines. Instances where this would theoretically move to a line off the screen (i.e., current line + n >24) cause a roll-up instead (see *****). The cursor is positioned at the beginning of the last line moved to (the bottom line of the screen if one or more roll-ups occurred).

If n=0, the cursor returns to the beginning of the current line.

If n<0, the cursor moves up n lines. A move to a line off (above) the screen causes a roll-down instead. The cursor is positioned at the beginning of the last line moved to the top line of the screen if 1 or more roll-downs occurred.

7. <u>TAB(n)</u>

Printer: (n>0). The print position is advanced to column n of the current line. If the column has already been passed, the TAB is ignored.

Workstation: (n>0). The cursor is moved to column n of the current line, erasing passed-over characters (by overwriting them with space characters). If the column has been passed the TAB is ignored.

In either case, if the tab position is greater than the SELECTed line length (always 80 characters for the workstation), the print position is advanced to column 1 of the next line. If N is negative or zero, the TAB is ignored.

8. <u>COL(n)</u>

Like TAB, but does not erase any passed-over characters. (TAB and COL are equivalent for the printer since no characters can be erased.)

9. <u>AT(r, c[, [e]])</u>

Printer: Ignored

Workstation: AT(r,c[,[e]]) moves the cursor to row r, column c of the screen, and optionally erases e characters starting at (r,c). The following rules hold:

a. $1 \leq r \leq 24$.

b. $1 \leq c \leq 80$.

c. e>0; if e is greater than the number of characters from the cursor position to the end of the screen, only characters to the end of the screen are erased.

d. If e <u>and</u> the preceding comma are omitted, no erasure occurs. If e is omitted but the preceding comma is included, the rest of the screen is erased, starting from (r,c).

Note that AT, like COL, has <u>no</u> effect on passed-over characters.

PUT Statement

General Form:

$$\text{PUT}\left\{\begin{array}{l}\text{file-exp}\\ \text{alpha-receiver}\end{array}\right\}\ \text{[[,]USING line number],arg[,arg]...}$$

$$\left[\text{,DATA}\left\{\begin{array}{l}\text{GOTO}\\ \text{GOSUB}\end{array}\right\}\left\{\begin{array}{l}\text{line number}\\ \text{statement label}\end{array}\right\}\right]$$

where:

$$\text{arg} = \left\{\begin{array}{l}\text{expression}\\ \text{alpha-expression}\\ \text{array-designator}\end{array}\right\}$$

PUT inserts data into the record area or alpha-receiver USING the referenced Image or FMT, if specified, or using standard format.

PUT does not destroy values not explicitly overwritten. Data PUT into a record area may be written to the file by a subsequent WRITE or REWRITE statement.

The DATA exit is taken if a data conversion error occurs.

Examples:

```
SELECT #1 "EXAMPLE" CONSEC, RECSIZE=16
OPEN #1 EXTEND, FILE="EXAMPLE", LIBRARY="DATA",VOLUME="VOL444"
PUT#1,B$
```

NOTE:

PUT may be used to convert numeric data to a format acceptable to COBOL programs. See Appendix E, Numeric Data Format Compatability between VS BASIC and COBOL, for information on numeric data compatibility between BASIC and COBOL.

READ Statement

General Form:

```
READ receiver [,receiver]...
```

A READ statement causes the next available elements in a DATA list (values listed in DATA statements in the program) to be assigned sequentially to the receivers in the READ list. This process continues until all receivers in the READ list have received values or until the elements in the DATA list have been used up. Each receiver must reference the corresponding type of data or an error will result.

The READ and DATA statements must be used together. If a READ statement references more receivers than the number of elements in a data list, the system uses the next DATA statement in statement number sequence. If there are no more DATA statements in the program, an error occurs and the program is terminated.

The RESTORE statement can be used to reset the DATA list pointer, thus allowing values in a DATA list to be re-used (see RESTORE).

NOTE:

DATA statements may be entered any place in the program as long as they provide values in the correct order for the READ statements.

Syntax examples:

```
100 READ A,B,C
200 DATA 4,315,-3.98

100 READ A$,N,B1$(3)
200 DATA "ABCDE",27,"XYZ"

100 FOR I=1 TO 10
110 READ A(I)
120 NEXT I
  ....
200 DATA 7.2, 4.5, 6.921, 8, 4
210 DATA 11.2, 9.1, 6.4, 8.52, 27
```

READ File Statement

General Form:

```
                                                        ⎧ >  ⎫
READ file-exp    ⎡[,]HOLD⎤  [,]  ⎡⎧KEY[expl]           ⎨ >= ⎬  alpha-expl⎫⎤
                 ⎣       ⎦       ⎢⎨                    ⎩ =  ⎭            ⎬⎥
                                 ⎣⎩RECORD=exp2                           ⎭⎦

⎡⎡[,]USING    ⎧line number    ⎫⎤   ,arg[,arg]...⎤
⎣⎣            ⎩statement label⎭⎦                ⎦
⎡,EOD⎧GOTO ⎫  ⎧line number    ⎫⎤   ⎡,DATA⎧GOTO ⎫⎧line number    ⎫⎤
⎣    ⎩GOSUB⎭  ⎩statement label⎭⎦   ⎣     ⎩GOSUB⎭⎩statement label⎭⎦
```

where:

```
    HOLD =  hold record for REWRITE or DELETE.  The
            record is held exclusively if in SHARED
            mode, i.e., no other user may access the
            record until REWRITE, DELETE, or another
            READ HOLD is executed.

    expl =  alternate key number for keyed READ on
            alternate indexed file (primary key used if
            expl = 0 or omitted).

    alpha-expl =  indexed file key specifier; the first record
                  whose key satisfies the condition is read.
                  Only as many characters as specified in KEYLEN
                  are compared; if the alpha-exp is shorter
                  (defined length) than KEYLEN, only as many
                  characters as its length are compared.

          exp2 =  record number (from 1) for CONSEC files only.

    USING ⎧line#    ⎫=line number or statement label of FMT or Image
          ⎨statement⎬ statement describing the input data format.
          ⎩    label⎭
    arg = ⎧receiver        ⎫
          ⎩array-designator⎭

          Data is moved (and optionally converted) into
          consecutive receivers.

    EOD =  end-of-data or invalid key exit, overriding
           the SELECT EOD.

    DATA = data conversion error exit.
```

The READ (file) statement causes a record in a disk or tape file to be read. The file must have already been opened with an OPEN statement (see OPEN).

If neither KEY nor RECORD is specified, the next consecutive record is read (using the established reference key in the case of ALTERNATE INDEXED files, i.e., the last used in READ KEY).

If no arg list is present, the data are left unconverted in the record area, and are accessible only through GET.

If USING is omitted, data are assumed to be in internal format. (See Subsection 8.4.7, Data Representation in File I/O).

Example:

```
SELECT #1 "EXAMPLE" CONSEC, RECSIZE=16
OPEN #1 INPUT, FILE="EXAMPLE",LIBRARY="DATA" VOLUME="VOLUME"
READ #1,B$
```

REM[ARK] Statement

```
General Form:

REM[ARK] [text string]

where:
    text string = any characters of lanks (except colons;
                  a colon indicates the end of the
                  statement).
```

The REM statement is used at the discretion of the programmer to insert comments or explanatory remarks in his program. When the compiler encounters a REM statement, it ignores the remainder of the statement, but not necessarily the rest of the line, as the following examples (lines 210 and 300) show.

Syntax examples:

```
100 REM SUBROUTINE
210 REM FACTOR:  F=Y/(X+1)
220 REM THE NUMBER MUST BE LESS THAN 1
300 REM ---- :PRINT "ERROR":REM STOP:STOP
```

The statements after the colon in line 210 and after the first and third colons in line 300 will be executed.

Either REM or REMARK is an acceptable statement.

RESTORE Statement

```
General Form:

RESTORE[ {expression                                  } ]
       [ {LINE = {line number    } [, expression]     } ]
       [ {       {statement label}                    } ]
where:

  {line number    }= line number or statement label of a DATA statement
  {statement label}  in the program.  If omitted, the first DATA statement'
                     is used.
    1 ≤ expression ≤ total number of DATA items in the program,
      beginning at the given line, if specified.  If omitted,
      default = 1.
```

The RESTORE statement allows the repetitive use of DATA statement values by READ statements. When RESTORE is encountered, the system resets the DATA pointer to the specified DATA value. A subsequent READ statement will read data values beginning with the specified value.

When a RESTORE statement is encountered, the system resets the DATA pointer to the (expression) data value in the program, beginning either at the first DATA statement (if 'LINE =' is omitted) or at the DATA statement at the specified line number or statement label.

If 'expression' is omitted, the pointer is set to the first data value in the program or in (or beyond) the specified DATA statement.

Syntax examples:

```
100  RESTORE
200  RESTORE 5
300  RESTORE (X-Y)/2
400  RESTORE LINE = 100
500  RESTORE LINE = 100, 3
```

The following program, for example,

```
100 DATA 1,2,3
200 DIM A(1,10)
300 FOR I=1 TO 10
400 IF I > =6 THEN RESTORE LINE=700, 3
500 READ A(1,I)
600 NEXT I
700 DATA 4,5,6
800 MAT PRINT A;
```

produces the following output:

```
1  2  3  4  5  6  6  6  6  6
```

RETURN Statement

```
General Form:

RETURN
```

The RETURN statement is used in a subroutine to return processing of the program to the statement following the last executed GOSUB or GOSUB' statement.

If entry was made to a marked subroutine via a special function key on the keyboard, the RETURN statement will return control to the interrupted INPUT or STOP statement.

Repetitive entries to subroutines without executing a RETURN should not be done. Failure to return from these entries causes return information to be accumulated which can eventually cause a stack overflow and the premature termination of the program. (Also see RETURN CLEAR.)

Examples:

```
100 GOSUB 300
200 PRINT X:STOP
300 REM    THIS IS A SUBROUTINE
400 -
500 -
  - -
  - -
900 RETURN:REM  END OF SUBROUTINE

100 GOSUB'03(A,B$)
200 END
300 DEFFN'03(X,N$)
400 PRINT USING 500,X,N$
500 % COST = $#,###,###.## CODE = ####
600 RETURN
```

RETURN CLEAR Statement

```
General Form:

RETURN CLEAR [ALL]
```

Return Clear subroutine return-address information, generated by the last or all executed subroutine calls, from memory.

The RETURN CLEAR statement is a dummy RETURN statement. The RETURN CLEAR statement causes subroutine return address information from the last previously executed subroutine call to be removed from the internal tables; the program then continues at the statement following the RETURN CLEAR.

If RETURN CLEAR ALL is specified, all subroutine return information is removed from the program stack. No RETURN or RETURN CLEAR may be executed before a subsequent GOSUB or GOSUB'.

The RETURN CLEAR statement is used to avoid memory stack overflow when a program continually exits from subroutines without executing a RETURN. This is particularly useful when using the Program Function keys to control program execution (from either STOP or INPUT). When a Program Function key is used in this manner, a subroutine branch is made to the appropriate DEFFN' statement to continue execution.

A subsequently executed RETURN statement causes the STOP or INPUT statement to be repeated automatically. However, the user may wish to continue a program without returning to the STOP or INPUT. In this case, the RETURN CLEAR statement should be used to exit from the DEFFN' subroutine. Executing a RETURN CLEAR statement when not inside a subroutine will result in an error.

Syntax example:

```
100 DEFFN'15
200 RETURN CLEAR
```

REWRITE Statement

```
General Form:

REWRITE file-exp [[,]SIZE = exp][[,]MASK = alpha-expl]
   [    [[,]USING {line number    }] ,arg[,arg]...]
   [    [         {statement label}]              ]
    [ ,DATA {GOTO } {line number    } ]
    [       {GOSUB} {statement label} ]
where:

     USING {line#          } = line number or statement label of FMT or
           {statement label}   Image(%) describing the output format.

     arg = {num-expression  }
           {alpha-expression}
           {array-designator}
     DATA = data conversion error exit.
```

REWRITE is used to overwrite an existing record, which must have been read with the HOLD option.

If the arg list is omitted, the record is assumed to have already been formatted in the record area with a PUT statement.

If the arg list is present, it is converted value-by-value using the Image or FMT Statement, if specified. Otherwise, standard format is used.

If the file is not an INDEXED VAR[C] file, the rewritten record size will be the same as that of the overwritten record; SIZE and the implicit arg-list size are ignored.

If the file is an INDEXED VAR[C] file, the size of the rewritten record is determined in one of the following ways:

1. Record size = SIZE expression, if included.

2. Record size = resultant size of the formatted arg list, if specified (see WRITE).

3. If arg list omitted, the rewritten record will be the same size as the record it overwrites.

REWRITE is not allowed for CONSEC VARC files.

MASK is used to set the alternate key mask for alternate indexed files. See the explanation of the MASK function in Section 8.5, Intrinsic File I/O Function, for more information. If MASK is omitted, the alternate-key mask for the record is rewritten unchanged.

Examples:

```
100 REWRITE #1,SIZE=A,MASK=MASK(#2), USING 300,A$,B,C%     !
200 DATA GOTO 1000
300 FMT CH(20), PIC(##.#), PD(3)
```

RND Function

General Form:

RND(numeric expression)

The RND (random number) function is used to produce a pseudorandom number between 0 and 1. The term "pseudorandom" refers to the fact that BASIC cannot produce truly random numbers. Instead, it relies on an internal algorithm which uses the last random number to generate the next one. The resulting sequence (list) of values, though obviously not truly random, is scattered about in the range zero to one in such a manner as to appear statistically random; thus the term "pseudorandom."

There are three ways to use RND(exp), based on the value of the argument:

1. exp <0 or exp >1 -- This produces the next pseudorandom number in the list as produced by the internal algorithm . If this is the first use of RND in the program, the previous value is set by the compiler at compilation.

2. 0<exp <1 -- This simply returns exp as the result and resets the list to this value.

3. exp =0 -- This is similar to Option 2, but produces a number whose value is computed from the time of day when the RND is executed, rather than from a user or compiler specified value.

See Section 2.6 for more information on RND.

Examples:

```
100 A=RND(.5)
200 B=RND(2)
300 C=RND(1)
400 PRINT "A=";A,"B=";B,"C=";C
```

```
Result:  A=.5   B=.259780899273209   C=.298807370711264
```

ROTATE[C] Statement

General Form:

```
ROTATE[C] (alpha-receiver, numeric expresion)
```

where:
   $-8 \le$ numeric expression ≤ 8

This statement rotates bits in the given alpha-receiver. If expression <0, rotation is left to right. If expression >0, rotation is right to left. Bits which are moved past one end of the receiver will be moved to the other end of the receiver.

If C is not specified, rotation occurs for each byte in the receiver. If C is specified, the entire receiver is rotated.

ROTATE operates on the defined length of the alpha-receiver.

Examples:

```
100 DIM A$5
200 A$ = HEX(345678AD)
300 ROTATE (A$,4)
400 PRINT HEXOF(A$)
```

Result:

```
436587DA02
```

```
500 ROTATE C (A$,-8)
600 PRINT HEXOF (A$)
```

Result:

```
02436587DA, assuming the previous result
```

ROUND Function

```
General Form:

ROUND(exp,exp)
```

ROUND(X,N) is equivalent to the expression:

```
SGN(X)*(INT(ABS(X)*10**N%+0.5)/10**N%)
```

Its effect is to round off the value of X to the precision specified by N. If N is positive, X is rounded off to N decimal places. If N is negative, X is rounded off to the Nth place to the left of the decimal point. If N is not an integer, it is truncated. For example:

```
ROUND(123.4567,4) = 123.4567
ROUND(123.4567,3) = 123.4570
ROUND(123.4567,2) = 123.4600
ROUND(123.4567,1) = 123.5000
ROUND(123.4567,0) = 123.0000
ROUND(123.4567,-1) = 120.0000
ROUND(123.4567,-2) = 100.0000
ROUND(123.4567,-3) =    0   etc.
```

Note that "rounding upward" occurs, unlike the INT function; when rounding 4.7 to 0 decimal places, it produces 5, not 4.

SEARCH Statement

```
General Form:
                                 ⎧<  ⎫
                                 ⎪<= ⎪
SEARCH [-] alpha-expl           ⎨>  ⎬ alpha-exp2
                                 ⎪>= ⎪
                                 ⎪<> ⎪
                                 ⎩=  ⎭

TO ⎧numeric array-designator   [STEP numeric expression]⎫
   ⎩ alpha-receiver                                      ⎭
```

SEARCH searches alpha-expl (defined length) for substrings of the same length as alpha-exp2 (actual length) satisfying the given relation.

If "-" is not specified, the SEARCH begins with the substring starting at the leftmost byte (byte 1) of alpha-expl; each subsequent substring checked has starting byte n bytes to the right of the previous substring, where n is the value of the STEP expression.

If "-" is specified, the SEARCH begins with the rightmost substring, i.e., starting at the (defined length alpha-expl minus actual length alpha-exp2 +1)th byte of alpha-expl. Subsequent substrings have starting byte n bytes to the left of that of the previous substring.

If STEP is omitted, n=1 and all substrings will be checked.

SEARCH terminates when it runs out of substrings of the proper length or reaches the limit of the TO argument. If expl is initially too short, no substring is checked.

Upon completion, the TO argument will contain the starting positions of the substrings found (from 1) in one of the following formats:

1. If "numeric array-designator", the array will contain the numeric starting positions in the order in which they were found. The first unused array element (if any) will contain 0. Any other unused elements remain unchanged.

2. If "alpha-receiver", each pair of bytes will contain the 2-byte binary representation of the starting positions, as with Option 1. The first unused pair of bytes (if any) will contain binary 0. Any other unused bytes remain unchanged.

In either Case 1 or 2, if the array or receiver is too short to contain all positions found, remaining ones are lost.

Example:

```
100 DIM A$40, N(1,8)
200 A$="SESSIONS OF SWEET SILENT THOUGHT"
300 SEARCH A$=STR(A$,1,1) TO N()
400 SEARCH -A$=STR(A$,1,1) TO B$
500 PRINT HEXOF(B$)
600 MAT PRINT N
```

Output:

```
0013000D000800040003000100002020
 1      3       4       8
 13     19      0       0
```

SELECT Statement

```
General Form:

SELECT select-elt [,select-elt][...]

where:
    select-elt = { PAUSE[int]                                       }
                 { RADIANS                                          }
                 { DEGREES                                          }
                 { GRADS                                            }
                 { PRINTER [(exp)]                                  }
                 { CRT                                              }
                 { WS                                               }
                 { POOL file number[,file number]...,BLOCKS=int     }

    PAUSE[int]= [int]/10second execution pause after each write to the work-
                 station.  If d=0 or omitted, no pause.  System default = no
                 pause.

   {RADIANS}   trig arguments/results in radians, degrees or grads,
   {DEGREES} = respectively.  (360 degrees = 2  radians = 400 grads.)
   {GRADS  }   System default = radians.

   {PRINTER} = route PRINT'ed output (PRINT, PRINTUSING, etc.) to the
   {CRT    }   line printer or workstation, as specified.  If no SELECT
   {WS     }   has been executed, such output is routed to the work-
               station by default.

              exp may be used following printer to specify non-standard
              printer line width, where

                          1 ≤ exp ≤ 162

              (if omitted or invalid, default = 132)

    POOL      = a buffer pool for the specified files.  (Files must be
                indexed.)

    BLOCKS    = the number of 2048 byte buffers in the pool.

    int       = an integer from 1 to 255.
```

A POOL specification can only appear after the SELECT File statements for the pooled files, and a particular file-number can only be included in a single POOL. Only indexed files OPENed in INPUT or IO modes can be pooled. Otherwise, this statement may be used anywhere and as often as desired. The select-elt's are processed one at a time, left to right.

Example:

```
100 SELECT PAUSE 9,PRINTER,DEGREES,POOL#1,#2,BLOCKS=2
```

SELECT File Statement

```
General Form:

SELECT file-number [,] "prname"[,] (Consecutive)   [,IOERR exit]
                                    {Indexed    }
                                    {Tape       }
                                    (Printer    )

where:
    File-number = #n, where n is an integer from 1 to 64.

    prname = 1-8 characters (alphanumeric, including @,#,$

    Consecutive = [VAR[C][,]] CONSEC, RECSIZE = int1[,EOD exit]

    Indexed = [VAR[C][,]] INDEXED, RECSIZE = int1, KEYPOS = int2,
          KEYLEN = int3 [,{ALTERNATE} alt-spec[,alt-spec...]] [,EOD exit]
                          {ALT      }

    alt-spec = KEY int4, KEYPOS = int5, KEYLEN = int6 [,DUP]
        int4 = 1 to 16, may not be repeated.
                                [(IL)]
    tape = [VAR[C][,]] TAPE,    [{NL}], RECSIZE = int7, BLKSIZE = int8,
                                [(AL)]
           DENSITY = { 800}[,EOD exit]
                     {1600}

    printer = PRINTER, RECSIZE = int10

    exit = {GOTO }{line number     }
           {GOSUB}{statement label }

    IL = IBM Labelled Tapes
    NL = Non Labelled Tapes
    AL = ANSI Labelled Tapes
```

SELECT file specifies the characteristics of a file which is to be opened (see the OPEN statement) and read from and/or written to (see READ, WRITE, REWRITE, GET, PUT, DELETE, and SKIP).

SELECT can specify four types of files:

Consecutive disk files -- Files which can only be read or written to sequentially. READ, WRITE, REWRITE, GET, PUT, and SKIP may be used.

Indexed disk files -- File indexed via a key field. The key length and position must be specified. Alternate keys may also be specified. Records can be accessed sequentially or by a specific key. READ, WRITE, REWRITE, GET, PUT, DELETE, and SKIP may be used.

Tape -- Files may be read from or written to a tape. READ, WRITE, GET, PUT, and SKIP may be used.

Printer -- Files may be written for output to the printer. The first two bytes in each record must be printer control characters (see VS Principles of Operation). Only WRITE and PUT may be used, and only OUTPUT mode can be used in the OPEN statement.

The SELECT statement sets up a user file block (UFB) of file information and a record area for the specified consecutive, indexed, tape, or printer file, referenced by the file number, with the supplied parameters used to set initial values in the UFB.

A file number may appear in at most one SELECT statement. All SELECTs must appear before any file I/O statements in the program.

file-number -- Pound-sign (#) followed by an integer from 1 to 64 inclusive. This file-number is used in all other I/O statements to refer to the file specified by this SELECT statement.

prname -- Literal string consisting of 1-8 alphabetic or numeric characters including $, #, and @). This is the external name used by the operating system to access the file and to prompt the user for file information.

VAR[C] -- Variable-length [optionally compressed] records. Neither VAR nor C need be set for any existing file, but they must be set for a file to be created (output mode) with variable-length (or compressed) records.

RECSIZE -- Record size for fixed-length files; maximum record size for variable-length files.

Limits:

CONSEC	$1 \leq intl \leq 2048$
VAR CONSEC	$1 \leq intl \leq 2024$
INDEXED	$1 \leq intl \leq 2040$
VAR INDEXED	$1 \leq intl \leq 2024$

KEYPOS -- Key position in record (from 1) for indexed files.

KEYLEN -- Key length (maximum = 255) for indexed files.

IOERR -- Branch taken if I/O error occurs on the selected file.

EOD -- Branch taken if end-of-data, invalid key or duplicate key on an I/O operation not having an EOD exit of its own.

ALTERNATE KEY, KEYPOS, KEYLEN, DUP (Duplicate Key values allowed) -- Key number, position, and length for one alternate key. This applies to Indexed files which allow (up to 16) alternate key access paths. For an existing file, the ALTERNATE key list may be either omitted or a subset of the existing alternate key structure. The key numbers specified must be identical to those used when creating the file. Alternate keys which are not included are not accessible by either READ or the KEY() function.

Syntax examples:

```
100 SELECT #1,"HEAP",VAR,CONSEC,RECSIZE=100,EOD GOTO 1000  !
200 IOERR GOSUB 200

300 SELECT#2,"OF",CONSEC.RECSIZE=50

400 SELECT#3,"BROKEN",INDEXED,RECSIZE=200,KEYPOS=1,KEYLEN= !
500 10,ALT KEY1,KEYPOS=11,KEYLEN=10,KEY2,KEYPOS=21,KEYLEN=10

600 SELECT#4"IMAGES",VAR,TAPE,NL,RECSIZE=15, BLKSIZE=1000  !
700 DENSITY=1600,EOD GOSUB 1000

800 SELECT#5,"WHERE",PRINTER,RECSIZE=134
```

SGN Function

General Form:

```
SGN(numeric expression)
```

The SGN function returns an integer value equal to -1 if the argument is less than zero, 0 if the argument equals zero, or +1 if the argument is greater than zero.

SIN Function

General Form:

```
SIN(numeric expression)
```

The SIN function returns a floating-point value that is the sine of the numeric expression specified as its argument. The expression is considered to be in units of radians, degrees, or grads, depending on the trig mode specified by the most recently executed SELECT statement. If no SELECT statement has been executed in the program or subprogram, the default mode is radians.

SIZE Function

```
General Form:

SIZE (file expression)
```

SIZE returns the size in bytes of the last record read from the specified file. The result is an integer.

SKIP Statement

```
General Form:

SKIP file-exp{[,]BEG    }  [,EOD{GOTO } {line number    }]
             {,num-exp  }       {GOSUB} {statement label}

where:

    num-exp = number of records to skip; forward if n>0;
              backward if n < 0.
        BEG = skip to beginning of file.
```

SKIP positions a CONSEC file forward or backward a number of records or to the beginning (BEG) of the file. The EOD exit is taken if a SKIP results in a position before the beginning or past the end of the file.

For example, if record 1 was just read, SKIP#n,2 will cause the next record read to be record 4. SKIP #n,-1 causes the same record to be reread by the next READ or GET statement. A SKIP value of 0 is effectively ignored.

Syntax examples:

```
100 SKIP #A,BEG
200 SKIP #1,B,EOD GOTO 1000
```

SQR Function

General Form:

```
SQR(numeric expression)
```

The SQR function returns a floating-point value that is the square root of the numeric expression specified as its argument.

STOP Statement

General Form:

STOP [alpha-expression]

The STOP statement interrupts program execution. When STOP is encountered, the word STOP followed by the given alpha-expression is printed at the workstation.

Execution may be continued in either of two ways:

1. ENTER continues execution at the next executable statement following the STOP statement.

2. Depressing a PF key corresponding to a marked subroutine causes the program to continue at the entry point of the subroutine. A corresponding RETURN will cause the STOP to re-execute.

Note that the execution of STOP is exactly like that of INPUT with no arguments. This applies to the use of PF keys for DEFFN' strings and subroutine entry. Although data cannot be entered directly into a variable from STOP, data may be passed to the arguments of a DEFFN' subroutine.

Syntax examples:

```
100 STOP
200 STOP A$
300 STOP "TWAS BRILLIG AND THE SLITHEY TOVES"
```

STR Function

General Form:

```
STR( {alpha expression  } [,[s][,[n]]])
     {alpha array string}
```

where:

```
    s  =  starting character in sub-string (an
          expression) (1 if omitted); cannot be zero or negative.
    n  =  number of consecutive characters desired
          (an expression); cannot be zero or negative.
```

The string function, STR, specifies a substring of an alpha variable or array string. With it, a portion of an alpha value can be examined, extracted or changed. For example, the statement

```
100 B$=STR(A$,3,4),
```

sets the receiver B$ equal to the third, fourth, fifth, and sixth characters of A$.

If n is omitted, the remainder of the alpha value is used, including trailing spaces.

The STR function can also be used as a receiver on the left side of an assignment (LET) statement to assign a value to a substring. For example:

```
100 A$="ABCDEF"
200 STR(A$,4,3)="XYZ"
300 PRINT A$

Output:  ABCXYZ
```

If the STR function is used on the left side of an assignment (LET) statement, and the value to be received is shorter than the specified substring, the substring is filled with trailing spaces. In this case, the first argument of the STR function must be an alpha receiver.

SUB Statement

```
General Form:

SUB "name" [ [ADDR](arg[,arg]...)]
  .
  .
  .
statements in subroutine
  .
  .
  .

where:
    "name" = name of subroutine (1-8 alphabetic or
             numeric characters; 1st alphabetic,
             including @,#, and $).

    arg =   { alpha scalar variable   }
            { numeric scalar variable }
            { array-designator        }
            { file number             }
```

SUB defines a subroutine with (or without) an argument list. (The SUB statement and use of external subroutines are discussed in Section 6.5, External Subroutines.) Its logical end is signalled by an END statement, just as in a main program. The optional return code is ignored by the BASIC calling program. SUB must be the first statement, other than REM, in the program.

The name specified in the SUB statement need not be the same as the object file name. Subroutines must be linked to their calling program prior to run-time; a CALL statement in the calling program initiates a branch to the beginning of the subprogram.

The optional ADDR syntax specifies the type of address list which the SUB routine expects to be passed to it to locate the passed args. This is explained in more detail in Subsection 6.5.4, Passing Arguments to External Subroutines.

Generally, when dealing entirely with BASIC programs/subprograms, ADDR should not be used; it should usually be used if the BASIC SUBroutine is being called from a non-BASIC (e.g. COBOL) subroutine.

Variables and arrays local to the subroutine (i.e. not in the arg list) obey the usual rules. However, they are initialized only on the first subroutine call; on subsequent calls, they retain their previous values and dimensions.

The file number argument, used in file I/O statements, is logically replaced by the passed file number or file-expression when CALL is executed. The file number thus refers to SELECT and other I/O operations executed in the main program; dummy file numbers may not, therefore, appear in SELECT statements in the subroutine, i.e., when a file number is received as a parameter, a SELECT statement for that file number in the subroutine is not permitted.

However, local file numbers may be used to set up (SELECT) an I/O area local to the subroutine, independent of and inaccessible to the calling program.

Other arguments are passed as follows:

1. Non-ADDR Form -- All array args must be specified as to type (matrix, vector) for proper argument passing to occur. This may be done in either of 2 ways:

   a. In one or more DIM statements occurring before the use of any of the dummy arrays. The dimensions specified are of no significance; only the vector - matrix distinction is noted by the program.

   b. If not in a DIM statement, the array is assumed to be a matrix.

   Arrays and receivers are not physically moved; the subroutine receives pointers to their locations and dimensions. Thus, changed values and array dimensions (MAT REDIM) may be returned to the calling program.

   Expressions and alpha-expressions that are not receivers must be created in temporary locations by the calling program; otherwise, pointers to their locations (and lengths, for alphas) are passed to the subroutine as in (A). Although values may be changed in the subroutine, these new values are not accessible by the calling program.

   In either case, the defined dimensions and lengths received by the SUBroutine specify the maximum area, as in a DIM or COM. MAT REDIM may change these dimensions subject to the usual rules and, as indicated, these new dimensions are retained upon return to the calling program.

2. ADDR Form -- SUB passes pointers to the locations of the passed arguments. All array dimensions and alphanumeric lengths are as specified in the SUB program (or are the default values).

   Otherwise, ADDR works the same as the non-ADDR form. Specifically, any changes to the data are reflected in the calling program upon return from the subroutine.

However, note that MAT REDIM has no effect outside the subroutine, since the dimensioning information from the calling program is inaccessible to the called subprogram.

No subroutine dummy argument may have the same name as either another dummy argument of the same type (scalar/array) or a COM argument specified in the subroutine.

A subroutine may call other subroutines, but may not call itself.

A source file may contain exactly one module, either a program or a subroutine.

```
100 SUB "AND"
200 A$ = STR("THE DRY STONE NO",5,3)
300 PRINT A$' "SOUND OF WATER"
400 END
```

```
100 SUB "ONLY" ADDR(A$,B,BC(),#N)
200 IF A$ AND "THERE IS A SHADOW" THEN B=20
300 END
```

```
100 SUB "123456" (A$)
100 PRINT A$
300 END
```

TAN Function

General Form:

```
TAN(numeric expression)
```

The TAN function returns a floating-point value that is the tangent of the numeric expression specified as its argument. The expression is considered to be in units of radians, degrees, or grads, depending on the trig mode specified by the most recently executed SELECT statement. If no SELECT statement has been executed in the program or subprogram, the default mode is radians.

TIME Function

```
General Form:

TIME
```

TIME returns an 8-character string containing the current time (accurate to hundredths of a second) in the form HHMMSShh.

TITLE Compiler Directive

General Form:

TITLE (expression)

The TITLE statement is a compiler directive (see Subsection 2.4.2, Compiler Directives). A TITLE statement must be the only statement on a line. When a TITLE statement is encountered during compilation, the compiler skips to the top of the next page of output listing and titles that page with the expression in the TITLE statement. All subsequent pages of the listing will be printed with the specified title until another TITLE statement occurs in the program text.

Example:

```
100 TITLE PART I: VARIABLE INITIALIZATION SECTION
```

When this statement is encountered the title

```
PART I: VARIABLE INITIALIZATION SECTION
```

would appear at the top of the page of source listing.

TITLE statements may not be continued nor may they be used in a multiple statement line.

TRAN Statement

General Form:

TRAN (alpha-receiver, alpha-expression) [REPLACING]

TRAN translates (in place) the alpha-receiver, using the alpha-expression as a translate table or list.

The defined length of the alpha-receiver is translated left-to-right, one byte at a time, as follows:

1. The alpha-expression (translate table) is moved to a separate location; thus, it cannot be affected by the translation.

2. Each byte is translated, in one of the following ways:

   a. REPLACING specified: The alpha-expression is treated as a list of consecutive byte pairs, ending either at a HEX(2020) pair or at the end (last full byte pair) of the alpha-expression. The second byte of each pair is a "translate from" byte, and the first a "translate to" byte.

      The alpha-expression is searched from left-to-right until a "translate from" matching the subject byte is found. The subject byte is then changed to the corresponding "translate-to" character. If a matching byte is not found, the subject byte is not changed.

   b. REPLACING not specified: The alpha-expression is treated as a table of consecutive "translate to" bytes. The subject byte is changed to the (n+1)th byte in the table, where n is the hex value of the subject byte. If the alpha-expression has fewer than n+1 bytes, the subject byte is not changed.

Program example:

```
100 A$="JOHN"
200 B$=HEX(00010203)
300 TRAN(A$,"MJAORHYN")REPLACING
400 TRAN(B$,"ABCDEF")
500 PRINT A$,B$
```

Output:

```
MARY        ABCD
```

UNPACK Statement

General Form:

UNPACK PIC (image) alpha-expression TO

$$\left\{\begin{array}{l}\text{numeric array-designator}\\ \text{numeric variable}\end{array}\right\}\left[,\left\{\begin{array}{l}\text{numeric array-designator}\\ \text{numeric variable}\end{array}\right\}\ldots\right]$$

where:
    image = [±][#...][.][#...][↑↑↑↑] (at least 1 "#")

The UNPACK statement is used to unpack numeric data that was packed by a PACK statement. Starting at the beginning of the specified alphanumeric expression, packed numeric data is unpacked and converted to internal floating-point values, and stored into the specified numeric variables or arrays. The format of the packed data is specified by the image (see PACK); thus, the same image that was used to pack the data should be used in the UNPACK statement. An error results if more numeric values are attempted to be unpacked than can exist in the alphanumeric expression (defined length used).

Syntax examples:

```
100 UNPACK PIC (####)A$ TO X,Y,Z
200 UNPACK PIC (+#.##)STR(A$,4,2) TO X
300 UNPACK PIC (+#.##    )A$() TO N()
400 UNPACK PIC (######)A$() TO X,Y,N(),M()
```

Program example:

```
100 X=24:DIM A$3
200 PACK PIC (####)A$ FROM X
300 PRINT X
400 PRINT HEXOF (A$)
500 UNPACK PIC (####)A$ TO Y
600 PRINT A$,Y
```

Output:

```
24
002420
$          24
```

$UNPACK Statement

```
General Forms:

$PACK ⎡(⎡⎧D=⎫⎤alpha-exp)⎤ alpha-receiver FROM arg[,arg]...
      ⎣  ⎣⎩F=⎭⎦          ⎦

                [,DATA ⎧GOTO ⎫⎧line number    ⎫]
                [      ⎩GOSUB⎭⎩statement label⎭]

where:
    line number      = line number or statement label of
    statement label    data conversion error exit

    arg = ⎧expression                                   ⎫
          ⎨alpha-expression, EXCEPT alpha array string  ⎬
          ⎩array-designator                             ⎭

    $UNPACK ⎡(⎡⎧D=⎫⎤alpha-exp)⎤alpha-expression TO arg[,arg]...
            ⎣  ⎣⎩F=⎭⎦          ⎦

                [,DATA ⎧GOTO ⎫ ⎧line number    ⎫]
                [      ⎩GOSUB⎭ ⎩statement label⎭]

where:
    arg =  receiver, EXCEPT alpha array string
           array-designator
```

See $PACK for explanation of syntax.

VAL Function

```
General Form:

VAL(alpha-exp[,d])

where:
    d = 1,2,3,4 (default = 1)
```

The VAL function converts the first d characters of the specified alphanumeric value to an integer. (The VAL function is also discussed in Section 5.6, Numeric Functions with Alpha Arguments.) The VAL function is the inverse of the BIN Function. VAL can be used wherever numeric functions normally are used.

VAL is particularly useful for code conversion and table lookups, since the converted number can be used as a subscript to retrieve the corresponding code or data from an array, or codes or information from DATA statements.

Syntax examples:

```
100 X=VAL(A$)
200 PRINT VAL("A")
300 IF VAL(STR(A$,3,1) 80 THEN 100
400 Z=VAL(A$)*10-Y
```

WRITE Statement

```
General Form:

WRITE file-exp[[,]SIZE=exp][[,] MASK = alpha-expl]
     [[[,]USING {line number    } ],arg[,arg]...]
     [[         {statement label} ]              ]
     [,EOD{GOTO }{line number    }] [,DATA{GOTO }{line number     }]
     [    {GOSUB}{statement label}] [     {GOSUB}{statement number}]
where:

    SIZE =  record size for VAR files.

    MASK = 2 byte mask alternate index mask for alternate indexed
           files. (If only 1 byte, right-padded with HEX(00)).

    USING {line#          }= line number of Image or FMT describing the
          {statement label}  formatting to be used on the output data.

    If USING is omitted, internal format is used.

    arg = {expression      }
          {alpha-expression}
          {array-designator}

    EOD  =  duplicate-key exit; overrides the SELECT EOD.

    DATA =  data conversion error exit (formatting error).
```

WRITE writes the next sequential record to a CONSEC file (OUTPUT, EXTEND, or SHARED mode) or a keyed record to an INDEXED file (IO, OUTPUT or SHARED mode). (The WRITE statement is also discussed in Section 8.4, The File I/O Statements.)

If an arg list is present, the data is moved one value at a time, using the format specified by Image(%), FMT, or internal formatting. If an arg list is not present, the data are taken directly from the record area, where they have already (presumably) been formatted with a PUT statement.

For non-VAR[C] files, the record size is as specified in SELECT; the SIZE parameter is ignored. For VAR[C] files, the record size is determined in one of the following ways:

1. Record size = SIZE expression, if specified.

2. If an arg-list is present, record size = resulting formatted record size. If USING is omitted, the data is left in internal format, with record size = sum of individual sizes:

floating-point	= 8 bytes
integer	= 4 bytes
alphanumeric	= defined length

3. If no arg-list, then record size is identical to that of the last record read or written, if any, or to the maximum RECSIZE.

For alternate indexed files, MASK is used to set the alternate key mask for the record (see description of the MASK function in Section 8.5, Intrinsic File I/O Functions). If omitted, the current MASK is used.

<u>Syntax example</u>:

```
100 WRITE #N,SIZE=100,MASK=A$,EOD GOTO 1000,DATA GOTO 1200
```

XOR Statement

General Form:

[LET] alpha-receiver = [logical exp] XOR logical exp

logical exp -- see Section 5.7, Logical Expressions.

The XOR operator logically exclusive OR's two or more alphanumeric arguments.

If the operand (logical expression) is shorter than the receiver, the remaining characters of the receiver are left unchanged. If the operand is longer than the receiver, the operation stops when the receiver is filled. (See Section 5.7, Logical Expressions, for more information on logical expressions.)

Examples:

```
HEX(0000)=HEX(0F0F) XOR HEX(0F0F)
HEX(0FF0)=HEX(00FF) XOR HEX(0F0F)
```

APPENDIX A
VS BASIC RESERVED WORDS

"When I use a word, it means just what I choose it to mean-- neither more nor less."
-- H. Dumpty, Through the Looking-Glass

The following is a list of all VS BASIC reserved words. Reserved words have a specific meaning to the BASIC compiler as statement verbs and keywords, and cannot be used either as variable names or statement labels. Some words on this list are not documented in this manual, since they are reserved for features to be implemented in future versions of the BASIC compiler.

$PACK	$UNPACK	ABS	ACCEPT	ADD	ADDC
ADDR	AL	ALL	ALT	AND	ANY
ARCCOS	ARCSIN	ARCTAN	ASORT	AT	ATN
BEG	BELL	BI	BIN	BLANK	BLINK
BLKSIZE	BLOCKS	BOOL	BOOL0	BOOL1	BOOL2
BOOL3	BOOL4	BOOL5	BOOL6	BOOL7	BOOL8
BOOL9	BOOLA	BOOLB	BOOLC	BOOLD	BOOLE
BOOLF	BRIGHT	BY	CALL	CH	CHAR
CLEAR	CLOSE	COL	COM	COMMON	CON
CONSEC	CONSTANT	CONVERT	COPY	COS	CRT
CURSOR	DATA	DATE	DECIMAL	DEF	DEFAULT
DEFFN	DEFFN'	DEGREES	DELETE	DENSITY	DIM
DISPLAY	DO	DPACK	DSORT	DUP	EJECT
ELSE	END	ENTER	EOD	ERROR	EXP
EXTEND	FAC	FILE	FILESEQ	FILL	FL
FLOAT	FMT	FN	FN'	FOR	FORM
FR	FROM	FS	GET	GO	GOSUB
GOSUB'	GOTO	GRAD	HALT	HEX	HEXOF
HEXPACK	HEXPRINT	HEXUNPACK	HOLD	IDN	IF
IL	INDEXED	INIT	INPUT	INT	INTEGER
INTO	INV	IO	IOERR	IPACK	KEY
KEYLEN	KEYPOS	KEYS	LEN	LET	LGT
LIBRARY	LINE	LOG	LONG	MASK	MAT
MAX	MIN	MOD	NEG	NEXT	NL
NOALT	NODISPLAY	NOGETPARM	NOT	NUM	NUMERIC
OBJECT	ON	ONLY	OPEN	OR	OUTPUT
PACK	PAGE	PAUSE	PD	PI	PIC
POOL	PRINT	PRINTER	PROTECT	PUT	RADIANS
RANGE	READ	REAL	RECORD	RECSIZE	REDIM
REM	REMARK	REPEAT	REPLACING	RESTORE	RETURN

REWRITE	RND	ROTATE	ROTATEC	ROUND	SCREEN
SEARCH	SELECT	SGN	SHARED	SHORT	SIN
SIZE	SKIP	SOURCE	SPACE	SQR	STEP
STOP	STR	SUB	SUB'	TAB	TABLE
TAN	TAPE	THEN	TIME	TIMEOUT	TITLE
TO	TRACE	TRAN	TRN	UNDERLINE	UNPACK
UPPERCASE	USING	VAL	VALIDATE	VALUE	VAR
VARC	VOLUME	WINDOW	WRITE	WS	XOR
XX	ZD	ZER	ZERO		

APPENDIX B
VS BASIC COMPILER OPTIONS

The following options are provided by the VS BASIC compiler:

SOURCE

If SOURCE = YES, the compiler produces a source listing of the compiled program, with accompanying diagnostics. If SOURCE = NO, no source listing is produced. (Diagnostics are produced if either SOURCE, PMAP, XREF, or ERRLIST is specified.)

PMAP

If PMAP = YES, the compiler produces a PMAP (program map) for the compiled program. A PMAP contains the machine instructions generated by each BASIC verb, with the address of each instruction, as well as a map of the static area showing the values and locations of all data items. A PMAP consists of five basic columns:

column 1 - BASIC verbs and line numbers.
column 2 - Address and object code.
column 3 - Assembler instructions.
column 4 - Operands for instructions, hex codes for literals.
column 5 - Comments.

If the program contains common variables (i.e., listed in a COM statement), a map of the common area will follow the PMAP, beginning with *COMMON on a new page. In the common area map the columns serve the same purpose as in the PMAP, with the exception of the first column, which will contain only *COMMON at the beginning of the map.

If there is no common area, a map of the static area immediately follows the PMAP, beginning with the word STATIC in column 1 on a new page.

In the static area map, the columns serve the same purpose as in the PMAP, with the exception of the first column. This first column contains either *STATIC, indicating the address of the contents of the Static section follow, or *PGT (Program Global Table) indicating the address of the information in that table follow. The Static section contains variables, while the PGT contains subroutine addresses, miscellaneous constants, and the like.

XREF

The XREF (cross reference listing) consists of five parts;

1. A listing of line number references (column one) and the line numbers where they are referenced (following columns).

2. A listing of the variable names, their lengths (alpha only), and, for arrays, their dimensions (all in column one). Each variable is followed by the location of the variable's storage area (or, for arrays, the descriptor and data area) on the same line, and the line numbers which reference the variable (on succeeding lines).

3. A listing of user-defined functions and the line numbers which reference them.

4. A listing of BASIC functions referenced, and the line numbers where they are referenced.

5. A listing of DEF FN' subroutines contained within the program and the line numbers which reference them.

LOAD

If LOAD = YES, the compiler creates an object program in VS object program format, and stores it in an output file. If LOAD = NO, no object program is produced and the compiler does not display an output definition screen to name the output file.

SYMB

If SYMB = YES, the compiler inserts symbolic debug information in the object program. If SYMB = NO, this information is not inserted, and the symbolic debug facility cannot be used to debug the object program at run time.

SUBCHK

If SUBCHK = YES, the compiler generates special code which checks the ranges of subscripts during program execution, and causes a program cancellation (execution interruption) if a subscript exceeds its defined limit. Otherwise, no check is performed on subscripts during execution.

ERRLIST

If ERRLIST = YES, a separate listing of the compiler diagnostics is produced.

FLAG

FLAG specifies the lowest level of error severity which will cause the compiler to print a diagnostic message. Any error with a severity code less than the specified FLAG value will not produce a diagnostic message.

STOP

STOP specifies the lowest level of error severity which will cause the compiler to abort the compilation. Any error with a severity code greater than or equal to the specified STOP value will terminate the compilation (no object program is produced).

LINES

LINES set the number of lines per page for all compiler produced printouts.

APPENDIX C
NUMERIC DATA REPRESENTATION IN VS BASIC: HEXADECIMAL/DECIMAL CONVERSION ERRORS

VS BASIC stores and operates upon numeric data represented as hexadecimal (base 16) numbers. Since 16 is not a power of 10, non-integral quantities which can be precisely represented in one number system (decimal or hexadecimal) cannot always be precisely represented in the other. Some numbers represented as non-repeating decimals, for example, become repeating hexadecimals when converted to hex representation. VS BASIC's input/output routines (PRINT, ACCEPT, INPUT, etc.) automatically take care of the interconversions of numeric data between the decimal form shown in ASCII characters on the workstation or printer, and the hexadecimal form stored internally.

There are situations, however, where small errors unavoidably occur when converting data between the two number systems. These are typically situations where many calculations are performed which compound small errors, such as in some trigonometric functions. For example, it is true by definition that the arcsine of the sine of 90 degrees is 90 degrees. However, if we run the program

```
100 SELECT D            /* SET TRIG MODE TO DEGREES */
200 PRINT ARCSIN(SIN(90))
```

the result shown on the workstation is not 90, but 89.9999994771725. In programs which do many numerical operations, such small conversion errors can accumulate through intermediate stages of a calculation to give larger errors in the final result. For example, if the arcsine expression just mentioned were to be used in subsequent calculations, the "correct" value to be used would be 90 degrees. If the result of the ARCSIN function were simply used directly, its value would not be precisely 90. Thus, the final results of subsequent calculations would be slightly inaccurate; if further conversion errors occurred in these calculations, the results would be even more inaccurate.

Accumulation of conversion errors of this sort can be avoided by judicious use of the ROUND function. In most cases, it is sufficient to use the ROUND function to round the final result to the desired precision immediately before it is output. In situations where many complicated calculations are being done, and greater precision is required in the final result, the ROUND function may be used at several intermediate points in the calculation to ensure that intermediate values are not processed with more significant digits than are warranted by the application. This will prevent conversion errors in low-order digits from propagating into the high-order digits of the final result.

APPENDIX D
FLOATING-POINT AND INTEGER CALCULATIONS

INTRODUCTION

In VS BASIC, the programmer has a choice between two types of numeric formats: integer and floating-point. Each format has unique features and limitations which make it particularly suitable for some applications and unsuitable for others. The programmer should clearly understand the differences between the two formats in order to intelligently select the most appropriate format for each application. In general, integer arithmetic is somewhat faster than floating point and requires less storage space. Integer results are precise, but the range of integer values is limited, and fractions cannot be used at all. Floating-point arithmetic does permit operations with fractions (that is, digits to the right of the decimal point) and it supports a much wider range of values than integer format; under certain conditions, however, floating-point results may lose some degree of precision (this problem is discussed below). A third type of numeric data format, packed decimal, is not available in VS BASIC but is supported in COBOL. The packed decimal format is briefly discussed at the conclusion of this appendix.

INTEGER FORMAT

Integer calculations are precise and consistent for a limited range of values. On the VS the range is from -2,147,483,648 to 2,147,483,647. Except for the occurrence of integer overflow, the standard set of integer operations, including arithmetic and relational operators, produce the expected results and obey the standard set of mathematical laws such as those concerning the associativity and commutativity of operations. Thus, integer equality ('=') tests for exactly equal and it is easy to understand when two integers are exactly equal.

Integer variables should be used whenever precise results are required and the expected range of values falls within the limited range supported by VS BASIC.

FLOATING-POINT FORMAT

Integer format is also referred to as "fixed-point" format because the decimal point is assumed to be permanently fixed just to the right of the low-order digit. In order to represent fractions, however, it is necessary to be able to locate the decimal point any place within the number (hence the term, "floating-point"). For a floating-point number, therefore, two pieces of information are required, a "fraction" and an "exponent": the "fraction" represents the number itself and the "exponent" specifies its magnitude. (That is to say, the exponent specifies the location of the decimal point.)

For example, the decimal number 15,000 could be represented as 1.5 X 10^4, where 1.5 is the fraction and 10^4 is the exponent. For reasons of accuracy, range, and performance, the fraction component of a floating-point number is stored in hexadecimal (base 16), while the exponent is stored in binary. The range of values that can be represented in floating-point format is much greater than integer (5.4×10^{-79} to 7.2×10^{75}) but the precision of the value is limited to approximately 15 digits.

A floating-point value is frequently an approximation of the value to the right of the decimal point. This is because, in contrast to the case of integer values, there is not an exact one-to-one correspondence between decimal fractions and hexadecimal fractions. In other words, there are certain decimal fractions which cannot be represented exactly as hexadecimal fractions, no matter how many bits of precision are utilized. (This situation is not uncommon, even in ordinary decimal arithmetic; consider, for example, the problem of representing 1/3 as a decimal fraction.)

The fact that many decimal fractions don't have exact hexadecimal equivalents means that some loss of precision can result when a decimal fraction is converted to hexadecimal. For example, the perfectly harmless decimal fraction 0.2 has a hexadecimal equivalent which is the repeating fraction .33333. In most cases, the loss of precision has no effect at all on the final result of a floating-point operation. Sometimes, however, the accuracy of the result may be affected. In particular, loss of accuracy may result when an iterative series of calculations is performed using decimal numbers which don't have exact hex equivalents. In such cases, the initially insignificant loss of precision gradually accumulates over a series of operations until it becomes large enough to influence the end result.

A second situation in which an insignificant loss of precision may produce unexpected results involves the use of comparison operators. When two floating-point numbers are compared, the comparison is made on their <u>hexadecimal</u> values. For example, the statement IF A = B THEN PRINT C compares the contents (in hexadecimal) of A and B; if the two values aren't <u>exactly</u> equal, the comparison fails. Consider, for example, the following short program:

```
100 A = 12
200 B = (12/10) *10
300 PRINT "A = "; A, "B = "; B, "THEY ARE";
400 IF A = B THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"

Result:  A = 12     B = 12      THEY ARE NOT EQUAL
```

This seemingly anomalous result illustrates how a loss of precision which has no noticeable effect on the result of the computation can influence the results of other operations. In this case, the division on line 200 produces a result, 1.2, which is normal in decimal but which is a repeating fraction in hexadecimal (because, as noted above, the fraction .2 does not have an exact hexadecimal equivalent). Thus, when the quotient is multiplied by 10, the result in B is not <u>exactly</u> the hexadecimal equivalent of 12, but a very close approximation. Since the comparison at line 400 expects the values of A and B to be exactly equal, the condition is not met, and the ELSE statement is executed. However, because formatted output statements in BASIC (such as the PRINT at line 300) perform an implicit rounding of the result which compensates for the one-bit loss of precision, both A and B print as 12.

CONCLUSION

For most applications, the loss of precision suffered by floating-point operations is insignificant and is not a cause of concern to the programmer. In those cases where it is a problem, the programmer has several courses of action available.

In some cases, use of the ROUND function can reduce the problems associated with floating-point representation. However, since the definition of rounding (e.g. increase if the digit is greater than or equal to 5) is based on the assumption of a decimal number system, ROUND may also produce unexpected results when the digit being rounded is very close to but slightly less than 5. This is again due to the fact that the floating-point number is an approximation of the decimal number. For example, consider the following program:

```
100 C = 30.5 * 11.69
200 D = ROUND (C,2)
300 PRINT "C=": C, "D="; D

Result: C = 356.545   D = 356.54
```

The expected result in D would, of course, be 356.55.

For applications in which it is not feasible to use integer arithmetic, the programmer should consider using a COBOL subroutine to perform the necessary calculations. COBOL supports packed decimal format, a data format not available in BASIC. With packed decimal format, arithmetic operations are performed directly in decimal, with no conversion to binary. Thus, any loss of precision due to conversion from decimal to binary (or to hexadecimal) is avoided. See the VS COBOL Language Reference Manual for a discussion of the COBOL language and the data formats available.

APPENDIX E
NUMERIC DATA FORMAT COMPATABILITY BETWEEN VS BASIC AND COBOL

VS BASIC stores integer data as binary integers in four bytes (one full word) of memory, and floating-point data as hexadecimal fractions in eight bytes. Other languages, however, may use other formats for storing numeric data. In particular, COBOL stores integer data as half-word binary integers, and non-integer numeric data in packed decimal format. In packed decimal format, each decimal digit of a number is coded into 4 bits of storage; the sign of the number is indicated by a hexadecimal digit attached to the right (low-order end) of the number. The decimal point is not stored; its position is specified only upon input or output.

For most applications, the number representation schemes used by other languages are of no concern to the VS BASIC programmer. However, if a BASIC program and a program written in COBOL are to process any of the same data, this difference cannot be ignored. This situation arises if a BASIC program and a COBOL program access the same data in either of the following ways:

1. By a BASIC program reading a data file written by a COBOL program, or vice versa,

2. By passing arguments between a calling program and a subprogram when one is in BASIC and the other is in COBOL.

Numeric data to be transferred from a BASIC program to a COBOL program must first be converted to half-word integer or packed decimal format. Similarly, numeric data transferred from a COBOL program to a BASIC program must be converted from half-word integer or packed decimal to BASIC full-word integer or floating-point format before any numeric operations can be performed on them.

These conversions are most easily accomplished using the BI and PD data specification of the FMT statement. For example, to write the number 123.45 to a data file to be read later by a COBOL program, one could do the following:

```
2600 NUMBER = 123.45
2700 WRITE #1, USING PACKED_DECIMAL, NUMBER
2800 PACKED_DECIMAL:  FMT PD(5,2)
```

To read a value from a data file which was written by a COBOL program, one could do:

```
3300 READ #2, USING PACKED_DECIMAL, VALUE
```

Packed decimal numbers are stored as a series of decimal digits with no decimal point. When a packed decimal number is read from a file and converted to VS BASIC floating-point format, the decimal point is inserted at the point indicated by the PD specification. Consequently, one must know beforehand where the "implied" decimal point is, so that the PD specification can be written appropriately.

In the case of numeric data passed between BASIC and COBOL calling programs and subprograms, data can be converted between full-word integer or floating-point and half-word integer or packed decimal formats using the PUT and GET statements in conjunction with a FMT statement with a BI or PD specification. Note that half-word integer and packed decimal representations of numbers in VS BASIC must be stored in alpha receivers. For example, suppose a BASIC program calls a COBOL subroutine to perform some calculations using the integer variable OPTION% and the floating-point variables RATE, TIME, and DISTANCE. Before the CALL statement, the data can be converted to the appropriate COBOL formats (one halfword integer, and three packed-decimal numbers) by performing

```
5600 PUT OPTION$, USING PD_FORM1, OPTION%
5700 PUT RATE$, USING PD_FORM2, RATE
5800 PUT TIME$, USING PD_FORM2, TIME
5900 PUT DISTANCE$, USING PD_FORM2, DISTANCE
6000
6100 PD_FORM1:  FMT BI(2)
6200 PD_FORM2:  FMT PD(6,2)
6300
6400 CALL ADDR SUB "CRUNCH" (OPTION$, RATE$, TIME$, DISTANCE$)
```

Note that although the arguments passed by the calling program are in a format designated by BASIC as an alphanumeric format, they correspond internally to COBOL's half-word integer and packed decimal numeric format. After the subprogram ends and control returns to the calling program, any changes made to the argument values by the COBOL subprogram can be retrieved and used as numeric values by the BASIC program with a conversion routine like the following:

```
6500 GET OPTION$, USING PD_FORM1, OPTION%
6600 GET RATE$, USING PD_FORM2, RATE
6700 GET TIME$, USING PD_FORM2, TIME
6800 GET DISTANCE$, USING PD_FORM2, DISTANCE
```

The variables OPTION%, RATE, TIME, and DISTANCE will now reflect any changes made to these values by the COBOL subprogram.

Similarly, if a BASIC subprogram is called by a COBOL program, the numeric arguments from the COBOL program will be passed to the BASIC subprogram as halfword integers and/or packed decimal numbers. Since these formats can only be received in BASIC by alpha receivers, the parameters in the SUB statement of the BASIC subroutine must be alpha receivers. Before any numeric operations can be performed, the data must be converted (or unpacked) to VS BASIC integer and/or floating-point format(s). This is done using the GET statement as above, and FMT statements with the appropriate BI and PD specifications. If the subroutine is to pass any numeric data back to the calling program, they must first be converted back to halfword integer or packed decimal format by PUT statements using the appropriate BI and PD specifications in one or more FMT statements.

APPENDIX F
VS CHARACTER SET

NOTE: b_0 *always equals zero**.				b_1 →	0	0	0	0	1	1	1	1
				b_2 →	0	0	1	1	0	0	1	1
				b_3 →	0	1	0	1	0	1	0	1
				High-Order Digit →	0	1	2	3	4	5	6	7
b_4 ↓	b_5 ↓	b_6 ↓	b_7 ↓	Low-Order Digit ↓								
0	0	0	0	0		â	SP	0	@	P	∘	p
0	0	0	1	1	♦	ê	!	1	A	Q	a	q
0	0	1	0	2	▶	î	''	2	B	R	b	r
0	0	1	1	3	◀	ô	=	3	C	S	c	s
0	1	0	0	4	→	û	$	4	D	T	d	t
0	1	0	1	5	—	ä	%	5	E	U	e	u
0	1	1	0	6	|	ë	&	6	F	V	f	v
0	1	1	1	7	··	ï	,	7	G	W	g	w
1	0	0	0	8	/	ö	(	8	H	X	h	x
1	0	0	1	9	\	ü	)	9	I	Y	i	y
1	0	1	0	A	^	à	•	:	J	Z	j	z
1	0	1	1	B	■	è	+	;	K	[	k	$
1	1	0	0	C	! !	ù	'	<	L	\	l	£
1	1	0	1	D	↕	Ä	–	=	M	]	m	é
1	1	1	0	E	ß	Ö	.	>	N	↑	n	ç
1	1	1	1	F	¶	Ü	/	?	O	←	o	¢

*Bit combinations 10000000 through 11111111 are field attribute characters.

APPENDIX G
VS FIELD ATTRIBUTE CHARACTERS

Bright	Modify	All	No line	80
Bright	Modify	Uppercase	No line	81
Bright	Modify	Numeric	No line	82
Bright	Protect	All	No line	84
Bright	Protect	Uppercase	No line	85
Bright	Protect	Numeric	No line	86
Dim	Modify	All	No line	88
Dim	Modify	Uppercase	No line	89
Dim	Modify	Numeric	No line	8A
Dim	Protect	All	No line	8C
Dim	Protect	Uppercase	No line	8D
Dim	Protect	Numeric	No line	8E
Blink	Modify	All	No line	90
Blink	Modify	Uppercase	No line	91
Blink	Modify	Numeric	No line	92
Blink	Protect	All	No line	94
Blink	Protect	Uppercase	No line	95
Blink	Protect	Numeric	No line	96
Blank	Modify	All	No line	98
Blank	Modify	Uppercase	No line	99
Blank	Modify	Numeric	No line	9A
Blank	Protect	All	No line	9C
Blank	Protect	Uppercase	No line	9D
Blank	Protect	Numeric	No line	9E
Bright	Modify	All	Line	A0
Bright	Modify	Uppercase	Line	A1
Bright	Modify	Numeric	Line	A2
Bright	Protect	All	Line	A4
Bright	Protect	Uppercase	Line	A5
Bright	Protect	Numeric	Line	A6
Dim	Modify	All	Line	A8
Dim	Modify	Uppercase	Line	A9
Dim	Modify	Numeric	Line	AA
Dim	Protect	All	Line	AC
Dim	Protect	Uppercase	Line	AD
Dim	Protect	Numeric	Line	AE
Blink	Modify	All	Line	B0
Blink	Modify	Uppercase	Line	B1
Blink	Modify	Numeric	Line	B2
Blink	Protect	All	Line	B4
Blink	Protect	Uppercase	Line	B5
Blink	Protect	Numeric	Line	B6
Blank	Modify	All	Line	B8
Blank	Modify	Uppercase	Line	B9
Blank	Modify	Numeric	Line	BA
Blank	Protect	All	Line	BC
Blank	Protect	Uppercase	Line	BD
Blank	Protect	Numeric	Line	BE

APPENDIX H
ASCII COLLATING SEQUENCE

1.		(space)
2.	"	(quotation mark)
3.	$	(currency symbol)
4.	'	(apostrophe, single quotation mark)
5.	(	(left parenthesis)
6.	)	(right parenthesis)
7.	*	(asterisk)
8.	+	(plus symbol)
9.	,	(comma)
10.	-	(hyphen, minus symbol)
11.	.	(period, decimal point)
12.	/	(stroke, virgule, slash)
13-22.		0 through 9
23.	;	(semicolon)
24.	<	(less than)
25.	=	(equal sign)
26.	>	(greater than)
27-52.		A through Z

APPENDIX I
VS BASIC ERROR MESSAGES

The following is a list of VS BASIC error messages. VS BASIC error messages are designed to be self-explanatory and self-documenting. For more information regarding the cause or resolution of the error see the section reference next to the error message.

ERROR NUMBER	ERROR MESSAGE	SEVERITY RETURN CODE	SECTION REFERENCE
101	Line number contains invalid characters.	12	2.3
102	Invalid character found.	8	Appendix E
103	Line number is out of sequence.	8	2.3
104	Literal not completed.	8	3.4
105	Literal improperly placed within statement.	8	2.3, 3.4
106	Incorrect constant or delimite.	8	3.3
107	Constant improperly placed within statement.	8	2.3, 3.3
108	'Non-unique' line number specified.	4	2.3
109	Invalid identifier name.	8	6.2
110	Constant too long - Significant digits lost.	4	3.3
111	Literal too long - Truncated to 256 characters.	4	3.4
112	A character string of length zero is invalid - Single blank substituted.	4	3.4
113	Improperly formed picture constant.	8	3.2
114	Numeric constant too large -Set to largest possible number.	4	3.3
115	Numeric constant too small - Set to zero.	4	3.3
116	HEX literal must contain an even number of characters.	8	3.4
117	Invalid boolean function - Must be 0-9 or A-F.	8	5.7
118	Invalid HEX literal.	8	3.4
119	Numeric constant too large for INTEGER-Treated as FLOATING POINT.	8	3.3
120	Last line of file contains a continuation character - your source file may be damaged.	8	2.3
121	Invalid exponent found in floating point constant.	8	3.3
122	Variable name exceeds 64 characters in length - Truncated.	8	3.2

ERROR NUMBER	ERROR MESSAGE	SEVERITY RETURN CODE	SECTION REFERENCE
201	Expecting end of statement but xxx was found.	8	2.3
202	Expecting line number or label but xxx was found.	8	6.3
203	Expecting statement verb but xxx was found.	8	2.3
204	Expecting yyy but xxx was found.	8	2.3
205	Expecting numeric or alpha expression but xxx was found.	8	4.3
206	Expecting alpha expression but xxx was found.	8	5.4
207	Expecting alpha receiver but xxx was found.	8	5.4
208	Expecting numeric scalar variable but xxx was found.	8	3.3, 3.5
209	Expecting alpha scalar variable but xxx was found.	8	3.4, 3.5
210	Expecting numeric array designator but xxx was found.	8	3.5
211	Expecting alpha array designator but xxx was found.	8	3.5
212	Expecting GOTO or GOSUB but xxx was found.	8	6.1, 6.4
213	Expecting matrix function or array variable but xxx was found.	8	9.2
214	Expecting matrix operator but xxx was found.	8	9.2
215	Expecting array variable but xxx was found.	8	3.5
216	Expecting relational operator but xxx was found.	8	4.2, 5.2
217	Expecting numeric receiver or numeric array designator but xxx was found.	8	3.2, 3.5
218	Expecting numeric array but xxx was found.	8	3.5
219	Expecting alpha array but xxx was found.	8	3.5, 5.3
220	Expecting alpha array or alpha array designator but xxx was found.	8	3.5, 5.3
221	Expecting print delimiter but xxx was found.	8	7.2
222	Expecting 1, 2, 3, or 4 but xxx was found.	8	
223	Expecting numeric array or numeric array designator but xxx was found.	8	3.5
224	Expecting numeric or alpha array designator but xxx was found.	8	3.5, 5.3
225	Expecting matrix variable or matrix function but xxx was found.	8	9.2
226	Expecting literal but xxx was found.	8	3.4
227	Expecting hexadecimal digit but xxx was found.	8	3.4
228	Expecting numeric constant or literal but xxx was found.	8	3.3, 3.4
229	Expecting numeric constant but xxx was found.	8	3.3
230	Expecting image but xxx was found.	8	7.4
231	Expecting integer constant but xxx was found.	8	3.3

ERROR NUMBER	ERROR MESSAGE	SEVERITY RETURN CODE	SECTION REFERENCE
232	Expecting array variable or alpha scalar but xxx was found.	8	3.4, 3.5
233	Expecting alpha scalar or alpha array designator but xxx was found.	8	3.4, 3.5, 5.3
234	Expecting 1 or 2 but xxx was found.	8	
235	SUB statement may not be preceded by any statements other than comments.	8	6.5
236	xxx has been previously defined.	8	4.4
237	Line number does not precede xxx statement.	8	2.3
238	Expecting GOTO or GOSUB but xxx was found.	8	6.1, 6.4
239	Invalid file number - Valid range is 1 to 64.	8	8.3
240	Expecting prname literal but xxx was found.	8	8.3
241	Expecting IOERR or EOD but xxx was found.	8	8.3
242	Expecting DEGREES, GRADS, RADIANS, PAUSE, CRT, WS, POOL, PRINTER, or file expression but xxx was found.	8	8.3
243	The SELECT statement must precede any disk Input/Output statements.	8	8.3
244	Either equal sign is missing in a LET statement or this statement starts with an unrecognizable word.	8	2.3
245	Multiply defined parameter in OPEN statement.	4	8.3
246	Expecting DATA but xxx was found.	8	8.3
247	Expecting EOD but xxx was found.	8	8.3
248	Function previously defined.	8	4.4
249	Invalid number of arguments.	8	6.5
250	Expecting integer 0-255 but xxx was found.	8	8.3
251	Expecting a function but xxx was found.	8	4.4
252	Expecting OPEN mode indication (INPUT, OUTPUT, IO, SHARED, or EXTEND) but xxx was found.	8	8.3
253	Invalid argument type in SUB statement.	8	6.5
254	Expecting format statement specification but xxx was found.	8	7.4
255	Expecting file expression but xxx was found.		
256	Expecting error specification but xxx was found.	8	8.3
257	Expecting file number or BLOCKS but xxx was found.	8	8.3
258	Expecting comma or equal sign but xxx was found.	8	2.3
259	This statement too long.	8	2.2
260	Expecting keyword option but xxx was found.	8	8.3
261	Missing comma before xxx.	6	2.3
262	Expecting equal sign or parenthesis after yyy but xxx was found.	8	2.3
263	Expecting CONSEC or INDEXED but xxx was found.	8	8.3

ERROR NUMBER	ERROR MESSAGE	SEVERITY RETURN CODE	SECTION REFERENCE
264	Invalid use of xxx in ACCEPT statement.	8	7.5
265	Incorrect number of subscripts.	8	3.5
266	Length must be in range 1 to 256.	6	8.3
267	xxx is not a valid name.	4	6.2
268	Invalid use of xxx is DISPLAY statement.	8	7.6
269	Nested IF statements are not allowed.	8	4.2, 5.2
270	xxx not yet implemented in ACCEPT or DISPLAY.	8	7.5, 7.6
271	xxx already specified in ACCEPT.	8	7.5
272	File was not previously specified in a SELECT statement.	8	8.3
273	Invalid device type for this function.	8	8.3
274	Invalid file attributes for this function.	8	8.3
275	File # xxx is already defined.	8	8.3
276	Invalid alternate key number - Must be in range 1 to 16.	8	8.3
277	Invalid alternate key for this file.	8	8.3
278	File does not have alternate indices.	8	8.3
279	Alternate index number already used.	8	8.3
280	Expecting TO, SUB, or SUB' but xxx was found.	8	6.5
281	Pause interval must be in range 1 to 255.	8	8.3
282	Expecting TO or SUB after GO but xxx was found.	8	6.3, 6.4
283	A field attribute character (FAC) may not immediately precede a literal.	8	7.5
284	Expecting comma or BEG but xxx was found.	8	8.3
285	Label xxx already exists.	8	2.3
286	Label yyy was previous by defined as a xxx.	8	2.3
287	Variable yyy was previously defined as a xxx.	8	3.3, 3.4
288	Function yyy was previously defined as a xxx.	8	4.4
289	Label xxx may not end with a $ or % character.	8	2.3
290	A string value may not be assigned to numeric receiver xxx.	8	3.2, 3.3
291	A numeric value may not be assigned to alpha receiver xxx.	8	3.2, 3.4
292	TRACE statement no longer supported - Use the symbolic debugger.	8	Appendix A
293	Null statement invalid after THEN or ELSE.	8	4.2, 5.2
294	yyy may not be declared as xxx variable.	8	3.2
295	Array dimension must be in the range 1 to 32767 - Default value of 10 used.	6	3.5
296	Format specifications must be greater than zero.	8	7.4
297	Expecting PIC but xxx was found.	8	7.5
298	The FILESEQ option is valid only for TAPE files.	8	8.3.2
401	Invalid operand in PRINT statement.	8	7.5

ERROR NUMBER	ERROR MESSAGE	SEVERITY RETURN CODE	SECTION REFERENCE
402	Compiler error.	12	1.4
403	Compiler error due to prior errors.	12	1.4
404	xxx invalid in USING list.	8	7.4
405	An invalid subroutine name or PRNAME has been corrected or replaced.	4	6.5, 8.3
406	Invalid OPEN option for this file access method.	8	8.3
407	Line number xxx missing before this line.	6	2.3
408	Constant invalid - Out of range.	8	3.3
409	SUB argument may not be declared in COMMON.	6	6.5
410	Invalid picture used in PACK, UNPACK, or CONVERT statement.	6	9.1
411	Invalid line length in SELECT statement.	4	8.3
412	FORM statement contains an invalid number in a FL or BI data specification.	6	2.3
413	Target line number is invalid for this type of statement.	8	6.4, 6.5
414	This file was not previously SELECTed.	4	8.3
415	File already specified in SELECT, POOL statement.	4	8.3
416	Code efficiency reduced due to complexity of expression.	4	4.3, 5.4
417	Invalid constant used as a subscript.	8	3.3, 3.4
418	Alternate key number must be in range 1 to 16.	8	8.3
419	Invalid alternate key specification.	8	8.3
420	File number must be within range 1 to 64.	8	8.3
421	This file has already been SELECTed.	8	8.3
422	Invalid record size specified.	8	8.3
423	The last character of the key is beyond the end of the record.	8	8.3
424	Target line number for EOD or DATA exit is invalid.	8	8.3
425	Integer matrix may not be the result operand of matrix inversion.	8	9.2
426	Prname truncated to eight characters.	4	8.3
427	Prname contains invalid character or starts with a digit.	8	8.3
428	Generated STATIC area too large.	12	1.4
429	Program too large to compile: xxx.	16	1.4
430	The selected file is not an indexed file - Buffer pooling may not be used.	4	8.3
431	Compiler error: xxx.	12	1.4
432	User function or routine is not defined.	6	4.4, 6.4
433	Generated COMMON area too large.	12	6.5
434	Combined COMMON and STATIC areas too large.	12	6.5
435	Array xxx too large.	12	3.5
436	Block size invalid.	4	8.3
437	Invalid tape density.	4	8.3

ERROR NUMBER	ERROR MESSAGE	SEVERITY RETURN CODE	SECTION REFERENCE
438	Line numbers greater than 65535 are incompatible with SYMBOLIC DEBUG - Use the EDITOR to renumber your program in smaller increments if you wish to use the symbolic debugger.	6	1.4, 2.3
439	Statement will cause run-time Stack overflow.	8	1.4
440	Statement too long.	12	2.2, 2.3
441	Label xxx missing.	6	2.3, 6.2
442	Primary key length may not exceed 255 characters.	8	8.3
443	Alternate key extends beyond the end of the record.	8	8.3
444	Alternate key length may not exceed 255 characters.	8	8.3
445	Alternate key number has already been specified for this file.	8	8.3
446	The sum of the lengths of the primary key and any alternate key may not exceed 255 characters.	8	8.3
447	xxx, yyy is referred to only once in this program. Check for possible spelling errors.	4	N/A

APPENDIX J
CVBASIC USER AID (CONVERSION from BASIC 2.3 to 3.2)

CVBASIC is provided as an aid to the BASIC programmer for converting from Version 2.3 of VS BASIC to Version 3.2 of VS BASIC. CVBASIC converts source code, single files or libraries, from Version 2.3 syntax to 3.2 syntax. Input to the utility must be syntactically correct source code (Version 2.3). Once the conversion program is complete, the output source file/library is created and an update listing is produced which indicates the success or failure of the conversion. This listing may be displayed at the workstation or printed. If the conversion was not successful, an error message is displayed which indicates the reason for the error and suggested corrective action. The output source code may then be compiled by the Version 3.2 BASIC compiler.

The new version of BASIC supports variable names up to 64 characters in length. CVBASIC accommodates this by inserting necessary spaces between the elements in the language, such as variables, reserved words and constants.

A summary of additional syntax changes which are automatically converted by CVBASIC include:

1. insert spaces around all VS BASIC reserved words;

2. convert SELECT D to SELECT DEGREES,
   SELECT R to SELECT RADIANS,
   SELECT G to SELECT GRADS,
   SELECT P to SELECT PAUSE;

3. convert #PI to PI;

4. convert CONVERT X to Y, (###) to
   CONVERT X to Y, PIC (###);

5. convert FMT PD (X.Y) to FMT PD (X,Y);

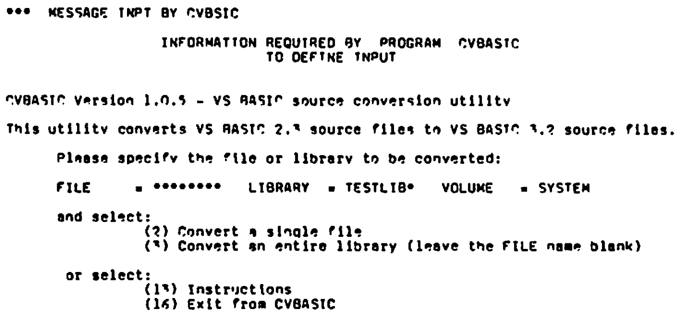
6. convert TRANS (A$,B$)R to TRANS(A$,B$)REPLACING;

7. convert PACK (###) TO PACK PIC (###);

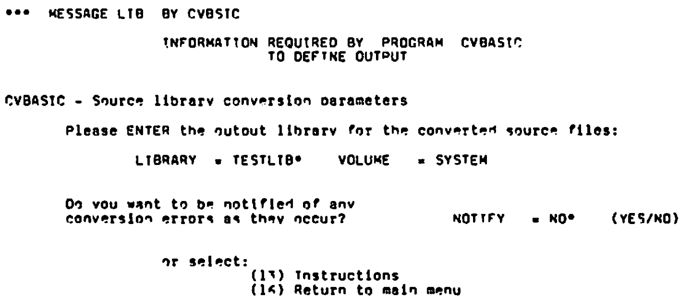
8. convert UNPACK (###) to UNPACK PIC (###);

USING CVBASIC

The screens in the figures below indicate the information needed to define the input and output of this program. Once CVBASIC is run successfully, the output source programs must then be compiled under Version 3.2 of the compiler. In addition, a listing of the files that were converted and any errors that occurred is produced. These error messages explain the condition that prevented the conversion and provide possible error correction solutions. If the library option is chosen and any of the files in the library are not valid source files, those files are automatically skipped. The names of the skipped files are written to the error listing.



```
***  MESSAGE INPT BY CVBSIC

                  INFORMATION REQUIRED BY  PROGRAM  CVBASIC
                              TO DEFINE INPUT


CVBASIC Version 1.0.5 - VS BASIC source conversion utility

This utility converts VS BASIC 2.3 source files to VS BASIC 3.2 source files.

     Please specify the file or library to be converted:

     FILE     = ********   LIBRARY  = TESTLIB*   VOLUME   = SYSTEM

     and select:
              (2) Convert a single file
              (3) Convert an entire library (leave the FILE name blank)

      or select:
              (13) Instructions
              (16) Exit from CVBASIC
```



```
***  MESSAGE LIB  BY CVBSIC

                  INFORMATION REQUIRED BY  PROGRAM  CVBASIC
                              TO DEFINE OUTPUT


CVBASIC - Source library conversion parameters

        Please ENTER the output library for the converted source files:

                  LIBRARY  = TESTLIB*    VOLUME   = SYSTEM


        Do you want to be notified of any
        conversion errors as they occur?          NOTIFY   = NO*    (YES/NO)


                  or select:
                             (13) Instructions
                             (16) Return to main menu
```

Information required to define input and output for CVBASIC

CVBASIC may be run from a procedure or from the Command Processor. The parameters for proceduralizing are listed in the table below. Conversions may be run as batch tasks only if they are fully parameterized (see <u>VS Procedure Language Reference</u> 800-1205PR).

PRNAME	Keyword	Length	Option(s)	Default	PF keys
(Main menu)					
INPUT	FILE	8		Blank	(2) Convert file
	LIBRARY	8		Source lib	(3) Convert library
	VOLUME	6		Source vol	(13) Instructions
					(16) Exit program
(File)					
OUTPUT	FILE	8		Output file	(13) Instructions
	LIBRARY	8		Output lib	(16) Main menu
	VOLUME	6		Output vol	
(Library)					
OUTPUT	LIBRARY	8		Output lib	(13) Instructions
	VOLUME	6		Output vol	(16) Main menu
	NOTIFY	3	YES/NO	NO	
(Output Listing)					
PRINT	FILE	8		Work name	(16) Main menu
	LIBRARY	8		Print lib	
	VOLUME	6		Print vol	
(If no conversion errors occurred)					
ERROR					(1) Main menu
					(11) Display results
					(15) Print results
					(16) Exit CVBASIC

PRNAME	Keyword	Length	Option(s)	Default	PF keys
(If conversion errors occurred)					
ERROR					(1) Main menu (11) Display errors (13) Instructions (15) Print errors (16) Exit CVBASIC
(Edit the file and renumber it)					
RENUMBER					(1) Skip this file (5) Renumber file (13) Instructions (16) Main menu

If CVBASIC is run interactively from a workstation, the name of the file undergoing conversion processing is displayed along with file size information. If the NOTIFY option is selected or just a single file is being converted, the user is notified, when the conversion is complete, of any errors that occurred. If the error can be corrected by renumbering the file, the user has the option of calling EDITOR to renumber, and then the conversion is automatically reattempted. If errors occur during processing, the error screen is displayed for the user to display or print the listing, return to the CVBASIC Main Menu, or exit from the program.

PROGRAM EXAMPLE

The following is a compilable program under Release 2.3 of VS BASIC that must be converted before running under Release 3.2 of VS BASIC:

```
100     SELECT D,P9
200     INPUTX
300     Y=SIN(X)
400     CONVERT Y TO Z$, (##.##)
500     PRINTZ$
600     GOTO200
```

When this program is used as the input program file for CVBASIC, the resulting file will be:

```
100     SELECT DEGREES, PAUSE 9
200     INPUT X
300     Y=SIN(X)
400     CONVERT Y TO Z$, PIC (##.##)
500     PRINT Z$
600     GO TO 200
```

Note that spaces have been provided, the abbreviated names have been expanded to their more self-documenting form, and the PIC clause has been added. These changes make long variable names possible and generally increase the clarity, readability and self-documenting nature of VS BASIC.

INDEX

ABS (Absolute Value) Function 119
ACCEPT Statement 85,86,88,90,120
   ALT Clause 90
   AT Clause 86
   CH Clause 87
   DATA Entry 88
   DATA Validation 88
   FAC Clause 87
   Fields 86
   Key Clause 89
   NO ALT Clause 90
   ON Key Clause 90
   PF Keys 89
   PIC Clause 87
   Positioning Data 86
   Range Clause 88
   Screen Formatting 86
ADD[C] 126
Addition 32
   Priority of 32
ADDR-type Subroutines 68,71,72
ALL 128
Alpha Array Strings 47
Alpha Expressions 48
Alpha Receivers 48
Alphanumeric 23,45
   Array 24
   Constants (see Literal Strings) 23
   Data 23
   Expressions 48
   Functions 49
   Literal Strings 23
   Operations 59
   Operators (see also Alphanumeric Data, Logical Operators with) 45
   Scalar 24
   TERMS 57
   Variables 24
Alphanumeric Data Formats 56
   Alphanumeric Length 56
   Defined Length 57
AND Logical Operator 129
ARCOOS (Arc Cosine) Function 130
ARCSIN (Arc Sine) Function 131
ARCTAN (Arc Tangent) Function 132
Argument in User-defined Functions 40
Arithmetic 114
Arrays 18
   Alphanumeric 24
   Array Strings 56
   Comparison between One- and Two-dimensional 27,28,29
   Default Dimensions 30
   Defining 18,24,26
   Dimensioning 29,115
   Elements of 21,26

Expressions in 26
Input Values for, Through INPUT Statement 26
Length of Elements 29
Naming 26
Numeric 44
One-dimensional 27
Subscripts 28
Two-dimensional 27
Array Assignment Statement 114
Operations with 114
Redimensioning Arrays with 114,115
Array Variables 26
Assignment Statement 114,188
ATN (arc tangent) Function 113

BASIC Character set 289
BASIC Compiler 8
LOAD 8
Options 8
SOURCE 8
SYMB 8
BASIC Statements 117
BIN Function 134
Blanks 14
In Image Statement 179
BOOLh Logical Operator 135
Branching 61
Program 61
Subroutine 62

CALL Statement 137
Character Set, BASIC 289
ASCII Collating Sequence of 291
Character String 25
CLOSE Statement 140
Closing Files (see CLOSE statement) 140
COM Statement 141
Comma 162
As an Insertion Character in FMT Statement 162
Use in INPUT Statement 182
Use in PRINT Statement 237
Comment 16
In Program 16,17
In REM Statement 16
In * Statement 16
Compiler 5
Basic 5
Editor 5
COMPILER OPTIONS 282
Computed GOSUB Statement 169
Concatenation (&) 46
Consecutive Files 94
Constants 18
Definition of 18

Control Specification, in FMT Statement
   (See also FOS, SKIP, and X) 161
Control Statements 61
   Branching 61
   Conditional 62
   Exit Conditions 62
   Halting Execution 61
   Looping 62
   Subroutine 62
   Unconditional 61
Control variable, in FOR Statement (see Index variable) 164
CONVERT Statement 142
COPY Statement 144
COS (Cosine) Function 145
Current Length 25
Current Value 25
Cursor 2

Data (see Alphanumeric and Numeric) 18
   Alphanumeric 23
   In FMT Statement 161
   In Image Statement 179
   In INPUT Statement 182
   In Scaler Assignment Statement (LET) 188
   Literals (Alphanumeric Constants) 23
   Numeric 19
Data Management System (DMS) 4
DATA Statement 146
   Relationship to READ Statement 146
   Relationship to RESTORE Statement 146
DATE Function 147
Decimal Point (.) ....
   In PRINT Statement 238
   Insertion Character in FMT Statement 162
DEF FN Statement 148
DEF FN' Statement 150
DEF Function Definition 148
Defining ....
   COM Statement 69,141
   DIM Statement 29,155
DELETE Statement 154
   Restriction with consecutive Files 154
Delimiters (see PRINT, INPUT) 228
DIM Function 38,156
DIM Statement 155
   Defining Alphanumeric Data with 155
   Defining Arrays with 155
DISPLAY Statement 91,157
DPACK Instruction 221

EDITOR 6
EJECT Compiler Directive 17,158
ELSE Keyword in IF Statement 177

END Statement 159
ERROR MESSAGES 293
EXP (Natural Exponential) Function 160
Expression, definition of 19
   Evaluation of 34
   Relational Operators in 34
External 67
   Argument Types 74
   Arguments 70
   Common Variables 72
   Compiling 69
   Form 68
   Initialization 73
   Linking 69
   Operation of 67
   Passing Values 69
   Running 69
   Use of 77

FAC (Field Attribute Character) 81
Field Format 229
Field Types 230
File Hierarchy 4
   File 4
   Volume 4
File I/O Buffering and Record Area 101
File I/O Modes 101
File Input/Output Statement 103
   Data Representation 103
   Errors 107
   Examples 109
   GET File 103
   PUT File 104
   READ File 103
   REWRITE File 104
   WRITE File 104
File Types 93
Filename, Definition of 5
FILESOEQ Instruction 221
Files 93
FLOATING POINT AND INTEGER CALCULATION 286
Floating Point Calculation 22,286
Floating-point Constant (E-format), definition of 19
Floating-point Data 41
FMT Statement 83,161
   Use of 82,161
   Used with Files 161
   Used with PRINTUSING 161
FOR Statement 164
FORM Statement 165
Format Control Specifications in FMT Statement
   (See also X, POS, SKIP) 82
FS Function 106,166

Functions ....
- ABS 119
- ALL 128
- ARCCOS 130
- ARCSIN 131
- ARCTAN 132
- ATN 133
- BIN 134
- COS 145
- DATE 147
- DIM 156
- EXP 160
- FS 106,166
- HEX 171
- INT 185
- KEY 186
- LEN 187
- LGT 190
- LOG 191
- MASK 192
- MATHEMATICAL 212
- MAX (Maximum Value) 211
- MIN (Minimum Value) 215
- MOD 216
- NUM 218
- Other Numerical 213
- PI 235
- POS 236
- RND 251
- ROUND 253
- SGN 260
- SIN 261
- SIZE 262
- SQR 264
- STR 266
- TAN 270
- TIME 271
- Trigonometric 212
- VAL 276

GET Statement 167
GO SUB Statement 64,168
GO SUB' Statement 65,169
GO TO Statement 170

HEXADECIMAL/DECIMAL CONVERSION ERRORS 285
Hexadecimal function (HEX) 171
HEXPACK Statement 172
HEXPRINT Statement 175
HEXUNPACK Statement 176

IF Statement 177
Image (%) Statement 83,179
   Use of 85,179
Indexed Files 94
   Alternate Keys 94
   Data Blocks 94
   Index Blocks 94
   Mask 94
   Primary Key 94
INIT Statement 181
INPUT Statement 79,182
   Compared to Workstation I/O Statements 80,182
Input/output Statements 78,79,114
   ACCEPT 79
   DISPLAY 78
   INPUT 79
   PRINT 78
INT (Integer) Function 185
Integer Calculation 22
Integer Constants 21
Integer data 42
Intrinsic functions 105
   FS 106
   Key 106
   MASK 107
   SIZE 107
IPACK Instruction........ 221

KEY Function 186

LEN (Length of Character String) Function 50,187
LET Statement 188
LGT (Logarithm to Base 10) Function........ 190
Library 4
LINE FORMAT 13
LINKER Utility 10
Literal, definition of 18
LOG (Logarithm to Base e) Function 191
Log On 3
Logical Expressions 53
   Evaluation of 54
Logical Operators 55

Machine Language 5
MASK Function 107,192
MAT ASSORT/DSORT........ 194
MAT CON........ 196
MAT IDN........ 198
MAT INPUT........ 199
MAT INV........ 201

MAT PRINT 204
MAT READ 205
MAT REDIM 206
MAT TRN 209
MAT ZER 210
MAT = ... 197
MAT + ... 193
MAT - 208
MAT * 203
MAT ()* 207
MATHEMATICAL FUNCTIONS 212
MAX (Maximum Value) Function 211
Menus 3
   Command Processor 3
MIN (Minimum Value) Function 215
Mixed Mode Arithmetic 41
Mixed Mode, definition of 19
MOD Function 216

NEXT Statement 217
NODISPLAY Instruction 221
NOGETPARM Instruction 221
NUM (Number of Numeric Characters in a Character String)
   Function 50,51,218
Number, definition of 18
Numeric ....
   Expressions 34
   Functions
      (see FUNCTIONS) 34,35,36,37,38,39
   Operators 31,32,33
Numeric Data 19
Numeric Functions with Alpha Arguments 50
   LEN 50
   NUM 51
   POS 51
   VAL 52
Numeric Terms 43
Numeric Variables 21
   Array 21
   Scalar 21

Object File 5
Object Program 5,11
   Running 11
ON Statement 219
One-dimensional Array 27,28
   Column Vectors 28
   Compared to Two-dimensional Array 27,28
   Lists 28
   Restrictions in Redimensioning 27,28
   Singly-Subscripted Arrays 28
   Vectors 28

OPEN Statement.. 220
Openations with ....
   Addition 114
   Identity function 114
   Inverse function 114
   Matrix multiplication 114
   Sort 114
   Subtraction 114
   Transpose function 114
Operators ....
   Arithmetic 32
   Assignment 31
   Assignment operator 45
   Concatenation operator 46
   Relational 33
   Relational operator 47
OR Logical Operator 224
Other Numerical Functions 213
Output 78
   DISPLAY Statement 78
   PRINT Statement 78

PACK Statment 225
PI Function 235
PIC 235
POS Function 50,51,236
PRINT Elements 238
Print Files 95, 140, 256
PRINT Statement 79,237
   Compared to Workstation and Printer I/O Statements 23
PRINT OUTPUT 79
   Expanded Print 80
   Line Feed 79
Program Development 5
Program Function Keys (PF) 2,66
PUT Statement 241

READ file statement 243
READ Statement 242
   Relationship to DATA Statement 242
   Relationship to RESTORE Statement
      (see RESTORE) 242
Receiver, definition of 19
Relational Operators.. 45
REM Statement... 245
RESERVED WORDS 13,280
RESTORE Statement 246
RETURN CLEAR Statement 248
Return Code 10
RETURN Statement 247
   Used with GOSUB 247

REWRITE Statement 249
  Restriction on Use 249
RND (Random Number) Function 38,251
ROTATE [C] Statement 252
ROUND Function 253

SEARCH Statement 254
SELECT File Statement 257
SELECT Statement 140,256
SGN (Signum) Function 260
SIN (Sine) Function 261
SIZE Function 105,262
SKIP File Control Statement 263
Sort 114
  Ascending 114
  Desending 114
Sort Statements 114
Source File 5
Source Program 5
Spaces, in Statements 14,200
SPACK Data Format 233
SPACK Statement 227
SQR (Square Root) Function 264
Statement, definition of 12
Statement Labels 62
Statements
  (see I/O Statement) 12,15,113-116
  Arithmetic 114
  Array Dimensioning 115
  Assignment Statement 113
  CALL 137
  CLOSE 140
  COM 141
  Continuation 15
  CONVERT 142
  COPY 144
  Data 146
  Data Conversion 113
  DEF 148
  DEF FN' 150
  Delete 154
  DISPLAY 157
  END 159
  FMT 161
  FOR 164
  FORM 165
  GET 167
  GOSUB 168
  GOSUB' 169
  GOTO 12,170

HEXPACK 172
HEXPRINT 175
HEXUNPACK 176
IF 12,177
IF THEN 12,177
IF ... THEN ... ELSE 177
Image (%) 179
INIT 181
INPUT 182
Input/Output Statements 114
LET 188
LET X=2 12
MAT ASORT/DSORT 194
MAT CON 196
MAT IDN 198
MAT INPUT 199
MAT INV 201
MAT PRINT 204
MAT READ 205
MAT REDIM 206
MAT TRN 209
MAT ZER 210
MAT = ... 197
MAT + ... 193
MAT - 208
MAT * 203
MAT ()* 207
MATRIX Statements 113
Multiple lines 15
NEXT 217
ON 219
OPEN 220
PACK 225
PRINT 237
PUT 241
READ 242
READ File 243
REM [ARK] 245
RESTORE 246
RETURN 12,247
RETURN CLEAR 247
REWRITE 249
ROTATE [C] 252
Rules 115
SEARCH 254
SELECT 256
SELECT File 257
SKIP 263
Sort 114
STOP 265
SUB 267
SUNPACK 275

TITLE 272
TRAN 273
UNPACK 274
WRITE 277
XOR 279
STOP Statement 265
STR (Portion of String) Function 266
SUB Statement 267
Subroutines 63,54,65
Definition of 62
External 67,68,69,70,72,73,74
GOSUB 64
GOSUB' 65
Internal 64
SUNPACK Data Format 233
SUNPACK Statement 227,275

TAN (Tangent) Function 270
THEN keyword in IF statement 177
TIME Function 271
TITLE Compiler Directive Statement 17,272
TRAN Statement 273
Trigonometric Functions 212
Two-dimensional Arrays 27
Double-subscripted Arrays 28
Matrices 28
Restriction in Redimensioning 27
Tables 28

UNPACK Statement 274

VAL Function 50,52,276
Variable 18
Definition of 18
VS Opeprating System 3
Variable na,e definition of 18

Workstation 2,80
WORKSTATION I/O 80
FAC (Field Attribute Characters) 81
Scrolling 81
Wraparound 80
WRITE File Statement 277

XOR Logical Operator Statement 279

2200 Disk Storage Format 232

To help us to provide you with the best manuals possible, please make your comments and suggestions concerning this publication on the form below. Then detach, fold, tape closed and mail to us. All comments and suggestions become the property of Wang Laboratories, Inc. For a reply, be sure to include your name and address. Your cooperation is appreciated.

800-1202BA-02

TITLE OF MANUAL **BASIC LANGUAGE REFERENCE**

COMMENTS:

Fold

Fold

(Please tape, Postal regulations prohibit the use of staples.)

WANG

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851

Attention: Technical Writing Department

Fold

Cut along dotted line.

Printed in U.S.A.
13-1019

International Representatives

Argentina
Bahamas
Bahrain
Bolivia
Botswana
Brazil
Canary Islands
Chile
Colombia
Costa Rica
Cyprus
Denmark
Dominican Republic
Ecuador
Egypt
El Salvador
Finland
Ghana
Greece
Guam
Guatemala
Haiti
Honduras
Iceland
India
Indonesia
Ireland
Israel
Italy
Ivory Coast
Japan
Jordan
Kenya
Korea
Kuwait
Lebanon
Liberia
Malaysia
Malta
Mexico
Morocco
New Guinea
Nicaragua
Nigeria
Norway
Paraguay
Peru
Philippines
Portugal
Qatar
Saudi Arabia
Scotland
Senegal
South Africa
Spain
Sri Lanka
Sudan
Tasmania
Thailand
Turkey
United Arab Emirates
Uruguay
Venezuela
Zimbabwe

United States

Alabama
Birmingham
Mobile

Alaska
Anchorage
Juneau

Arizona
Phoenix
Tucson

California
Anaheim
Burlingame
Culver City
Emeryville
Fountain Valley
Fresno
Los Angeles
Sacramento
San Diego
San Francisco
Santa Clara
Ventura

Colorado
Englewood

Connecticut
New Haven
Stamford
Wethersfield

District of Columbia
Washington

Florida
Coral Gables
Hialeah
Hollywood
Jacksonville
Miami
Orlando
Sarasota
Tampa

Georgia
Atlanta
Savannah

Hawaii
Honolulu
Maui

Idaho
Boise

Illinois
Arlington Heights
Chicago
Morton
Oakbrook
Park Ridge
Rock Island
Rosemont
Springfield

Indiana
Fort Wayne
Indianapolis
South Bend

Iowa
Ankeny

Kansas
Overland Park
Wichita

Kentucky
Louisville

Louisiana
Baton Rouge
Metairie

Maine
Portland

Maryland
Baltimore
Bethesda
Gaithersburg
Rockville

Massachusetts
Boston
Burlington
Chelmsford
Lawrence
Littleton
Lowell
Methuen
Tewksbury
Worcester

Michigan
Grand Rapids
Kalamazoo
Lansing
Southfield

Minnesota
Eden Prairie
Minneapolis

Mississippi
Jackson

Missouri
Creve Coeur
St. Louis

Nebraska
Omaha

Nevada
Las Vegas

New Hampshire
Manchester

New Jersey
Bloomfield
Clifton
Edison
Mountainside
Toms River

New Mexico
Albuquerque
Santa Fe

New York
Albany
Jericho
Lake Success
New York City
Rochester
Syossett
Syracuse
Tonawanda

North Carolina
Charlotte
Greensboro
Raleigh

Ohio
Akron
Cincinnati
Cleveland
Independence
Toledo
Worthington

Oklahoma
Oklahoma City
Tulsa

Oregon
Eugene
Portland
Salem

Pennsylvania
Allentown
Erie
Harrisburg
Philadelphia
Pittsburgh
State College
Wayne

Rhode Island
Providence

South Carolina
Charleston
Columbia

Tennessee
Chattanooga
Knoxville
Memphis
Nashville

Texas
Austin
Dallas
El Paso
Houston
San Antonio

Utah
Salt Lake City

Virginia
Newport News
Norfolk
Richmond
Rosslyn
Springfield

Washington
Richland
Seattle
Spokane

Wisconsin
Appleton
Brookfield
Green Bay
Madison
Wauwatosa

International Offices

Australia
Wang Computer Pty., Ltd.
Adelaide, S.A.
Brisbane, Qld.
Canberra, A.C.T.
Perth, W.A.
South Melbourne, Vic 3
Sydney, NSW

Austria
Wang Gesellschaft, m.b.H.
Vienna

Belgium
Wang Europe, S.A.
Brussels
Erpe-Mere

Canada
Wang Canada Ltd.
Burlington, Ontario
Burnaby, B.C.
Calgary, Alberta
Don Mills, Ontario
Edmonton, Alberta
Halifax, Nova Scotia
Hamilton, Ontario
Montreal, Quebec
Ottawa, Ontario
Quebec City, Quebec
Toronto, Ontario
Victoria, B.C.
Winnipeg, Manitoba

China
Wang Industrial Co., Ltd.
Taipei

Wang Laboratories, Ltd.
Taipei

France
Wang France S.A.R.L.
Paris
Bordeaux
Lille
Lyon
Marseilles
Nantes
Nice
Rouen
Strasbourg

Great Britain
Wang (U.K.) Ltd.
Richmond
Birmingham
London
Manchester

Hong Kong
Wang Pacific Ltd.
Hong Kong

Japan
Wang Computer Ltd.
Tokyo

Netherlands
Wang Nederland B.V.
IJsselstein
Groningen

New Zealand
Wang Computer Ltd.
Auckland
Christchurch
Wellington

Panama
Wang de Panama (CPEC) S.A.
Panama City

Puerto Rico
Wang Computadoras, Inc.
Hato Rey

Singapore
Wang Computer (Pte) Ltd.
Singapore

Sweden
Wang Skandinaviska AB
Stockholm
Gothenburg
Malmö

Switzerland
Wang A.G.
Zurich
Basel
Bern
Geneva
Lausanne
St. Gallen

Wang Trading A.G.
Zug

West Germany
Wang Deutschland, GmbH
Frankfurt
Berlin
Cologne
Düsseldorf
Essen
Freiburg
Hamburg
Hannover
Kassel
Mannheim
Munich
Nürnberg
Saarbrücken
Stuttgart

WANG

ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

Printed in U.S.A.
800-1202BA-03
6-81-5M