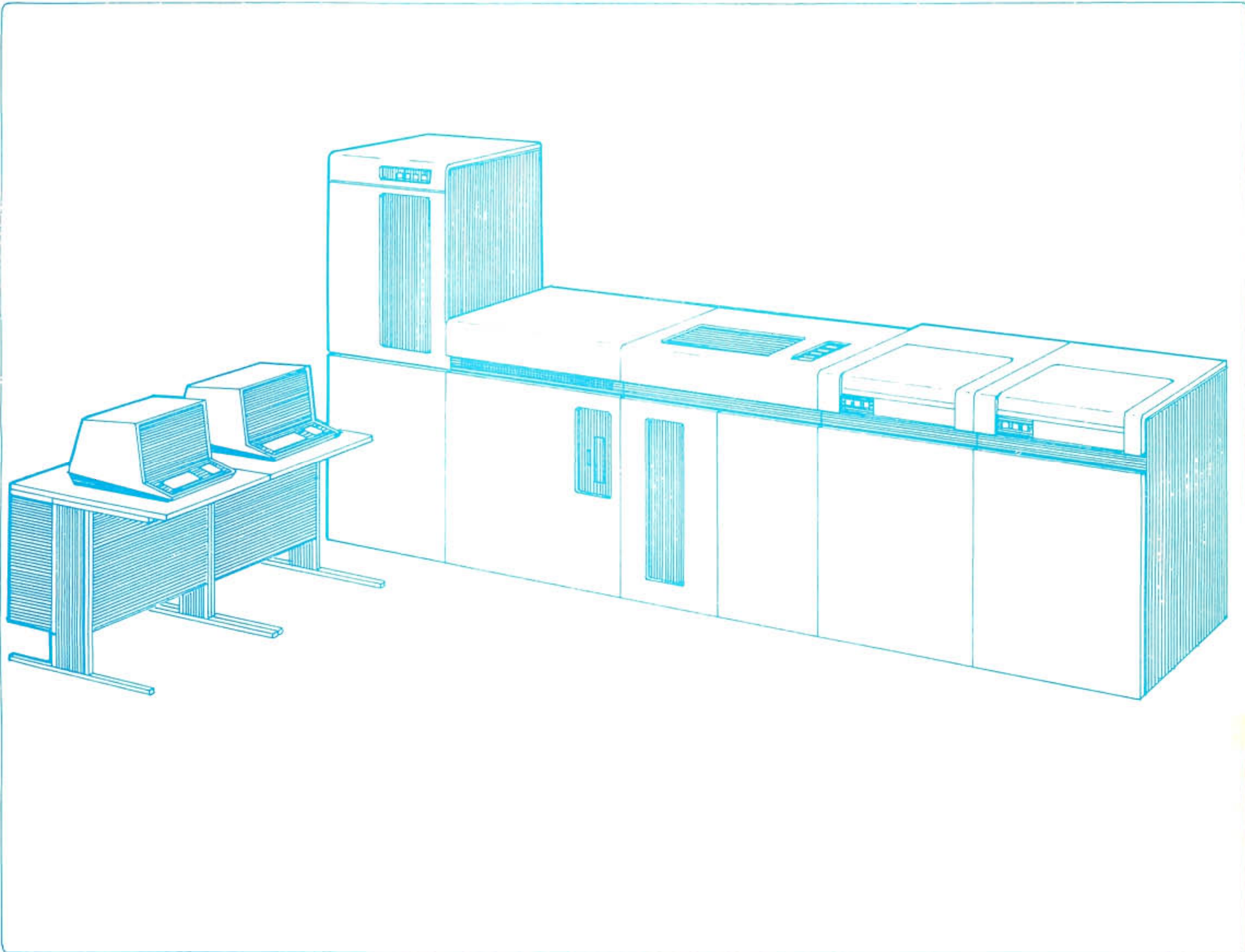
WANG

2200VS



2200VS BASIC Language Reference Manual

2200VS
BASIC Language Reference Manual

Release: 1
September 1978
© Wang Laboratories, Inc., 1978
800-1202BA-01

LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 851-4111, TWX 710 343-6769, TELEX 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase agreement, lease agreement, or rental agreement by which this equipment was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of this manual or any programs contained herein.

LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 851-4111, TWX 710 343-6769, TELEX 94-7421

TABLE OF CONTENTS

PAGE

PART I INTRODUCTION TO BASIC. 1

CHAPTER 1 INTRODUCTORY CONCEPTS. 2

1.1 An Overview: BASIC on the Wang VS 2
1.2 VS Hardware Configuration. 4
1.3 Entering and Compiling a BASIC Program . . . 4
1.4 Running the BASIC Object Program 11
1.5 File Types Supported on the VS 11
1.6 Program Development. 11
1.7 Source File Format 12
1.8 BASIC Language Structure 15
1.9 Rules of Syntax. 16

CHAPTER 2 NUMERICS 19

2.1 Introduction 19
2.2 Numeric Constants. 20
2.3 Numeric Variables. 21
2.4 Expressions. 26
2.5 Mixed Mode 27
2.6 Numeric Functions 27
2.7 Summary of Rules, Formats, and Syntax . . . 33

CHAPTER 3 ALPHANUMERICS. 36

3.1 Alphanumeric Character Strings 36
3.2 Alphanumeric String Variables. 37
3.3 Literals 40
3.4 Alpha Receivers and Alpha Expressions. . . . 41
3.5 Concatenation of Strings 43
3.6 Alpha Array Strings. 43
3.7 Hexadecimal Literal Strings. 44
3.8 Logical Expressions. 44
3.9 Functions with Alpha Arguments 47
3.10 Summary of Rules, Formats, and Syntax . . . 51

CHAPTER 4 CONTROL STATEMENTS 56

4.1 Introduction 56
4.2 Subroutines. 58
4.3 Internal Subroutines 59
4.4 External Subroutines 61

PAGE

CHAPTER 5 WORKSTATION AND PRINTER INPUT/OUTPUT STATEMENTS 69

5.1 Introduction 69
5.2 Printer Output 73
5.3 FMT and Image Statements 73
5.4 The Workstation Screen 75
5.5 Field Attribute Characters 76
5.6 DISPLAY. 77
5.7 ACCEPT 78

CHAPTER 6 TAPE AND PRINTER FILE I/O. 82

6.1 Introduction 82
6.2 File Hierarchy 83
6.3 Selecting File Numbers 84
6.4 Opening a File 86
6.5 Summary of I/O Statements. 87
6.6 File I/O System Functions. 89
6.7 Error Recovery 90

CHAPTER 7 SPECIAL STATEMENTS: MATRIX AND DATA CONVERSION STATEMENTS 93

7.1 Data Conversion Statements 93
7.2 Matrix Statements 94

PART II BASIC KEYWORD FORMATS. 98

ACCEPT 100
ADD[C] 106
ALL Function 108
AND Logical Operator 109
BIN Function 110
BOOLh Logical Operator 111
CALL 113
CLOSE. 116
COM. 117
CONVERT. 118
COPY 120
DATA 121
DATE Function. 122
DEFFN. 123
DEFFN' 125
DELETE 129
DIM. 130
DIM Function 131
DISPLAY. 132
END. 133
FMT. 134

PAGE

FN Function. 137
FOR. 138
FS Function 139
GET. 140
GOSUB. 141
GOSUB' 142
GOTO 143
HEX Literal String 144
HEXPACK. 145
HEXUNPACK. 148
IF...THEN...ELSE 149
Image (%). 151
INIT 153
INPUT. 154
KEY 157
LEN Function 158
LET. 159
MASK Function 161
MAT+ 162
MAT ASORT/DSORT 163
MAT CON. 164
MAT =. 165
MAT IDN. 166
MAT INPUT. 167
MAT INV. 169
MAT *. 171
MAT PRINT. 172
MAT READ 173
MAT REDIM. 175
MAT ()*. 176
MAT -. 177
MAT TRN. 178
MAT ZER. 179
NEXT 183
NUM Function 184
ON 185
OPEN 186
OR Logical Operator 190
POS Function 191
PRINT. 192
PRINTUSING 196
PUT. 197
READ 198
READ File 199
REM. 201
RESTORE. 202
RETURN 203
RETURN CLEAR 204
REWRITE. 205
RND 207

PAGE

ROTATE 208
ROUND 209
SEARCH 210
SELECT 212
SELECT File 213
SIZE 216
SKIP 217
STOP 218
STR Function 219
SUB. 221
TIME Function. 224
TRACE. 225
TRAN 227
VAL Function 228
WRITE. 229
XOR Logical Operator 231

APPENDIX A GLOSSARY 232

APPENDIX B 2200T and 2200VP Considerations. 236

APPENDIX C VS Field Attribute Characters. 250

APPENDIX D BASIC Compiler Options 251

APPENDIX E VS Character Set 254

APPENDIX F ASCII Collating Sequence 255

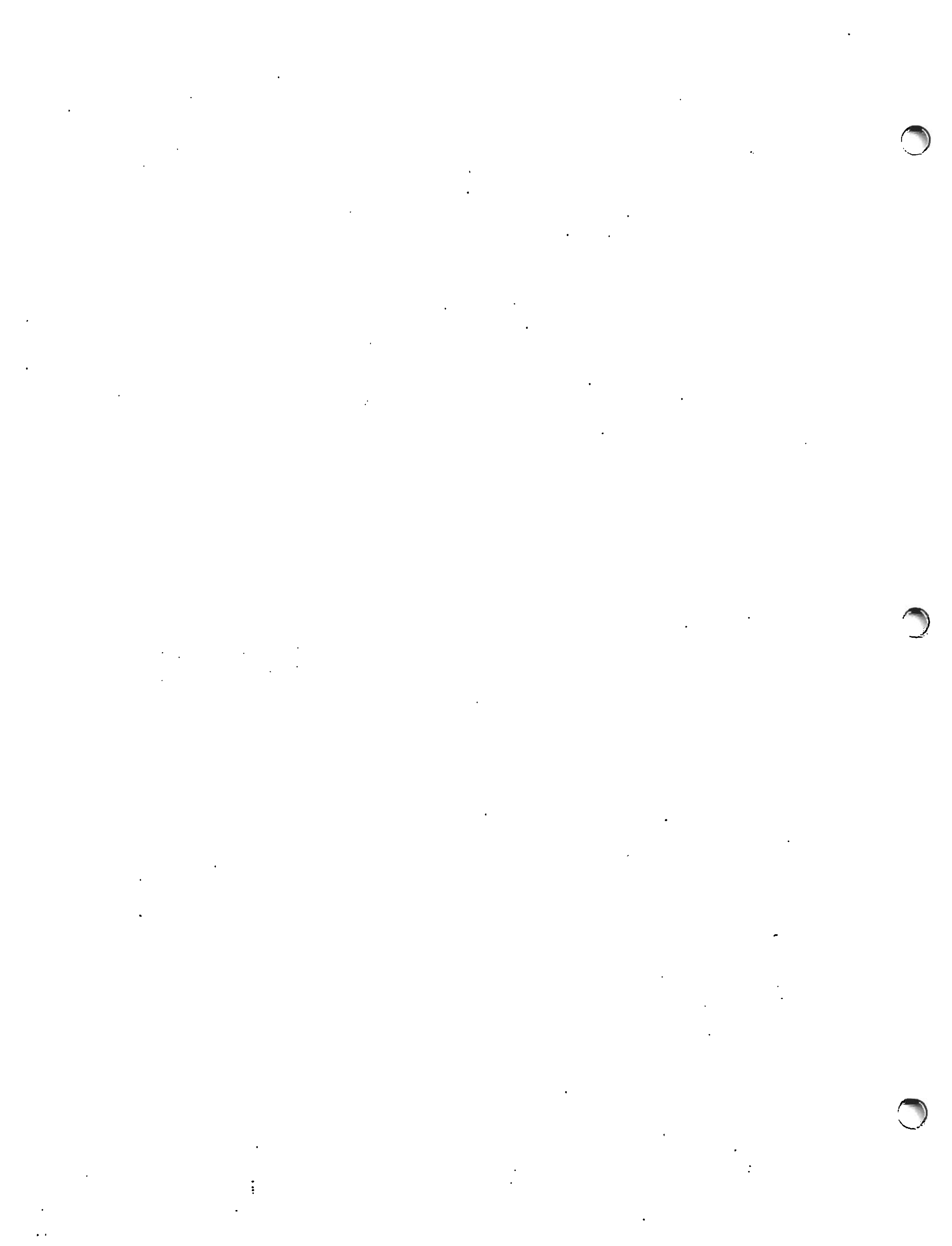
APPENDIX G BASIC Keywords 256

HOW TO USE THIS MANUAL

This manual is designed as a reference for Wang VS BASIC. For information on operating system programs and utilities, see the _2200VS Programmer's Introduction_.

The BASIC manual is divided into two parts. Part I, which contains Chapters 1 through 7, has general discussions of the parts of the language, while Part II contains short descriptions of each of the BASIC instructions, including details of the required formats.

VS BASIC is very similar to 2200 BASIC with the most important differences being in workstation and disk I/O. See Section 5.2 for information on workstation I/O, Chapter 6 for disk I/O information, and Appendix B for additional information on the 2200T and VP.



PART I

INTRODUCTION TO BASIC

CHAPTER 1 INTRODUCTORY CONCEPTS

1.1 AN OVERVIEW: BASIC ON THE WANG 2200VS

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 designed to be easily learned by beginning programmers. There are two reasons why this is so:

1. BASIC statements bear a close resemblance to English-language statements, giving beginning programmers clues to their 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 he or she has a specific need for those capabilities.

The advanced capabilities of VS BASIC are diverse. VS BASIC can perform computations using either floating-point or integer quantities. For scientific applications, a full set of functions - arithmetic, trigonometric, and transcendental - are provided. In addition, the programmer may define his own functions in the course of writing a program. Exponential notation is supported throughout the language. For applications in business, BASIC provides powerful editing features for producing formatted screen, printer, or disk output. For simulations and system programming, BASIC allows the programmer to perform a variety of conversions to and from binary, provides all 16 Boolean functions of two variables, and allows hexadecimal output. BASIC allows alphanumeric data ("strings" of text) to be manipulated as easily as numeric quantities. Strings can be concatenated (put together), broken up into substrings, or searched.

VS BASIC supports both numeric and alphanumeric arrays (groups of variables referred to by a single name). Numeric expressions (subscripts) select one of the elements of the array, allowing a single BASIC statement to access different variables on successive iterations. Both singly-subscripted variables (one-dimensional arrays, or vectors), and doubly-subscripted variables (two-dimensional arrays, or matrices) are supported. A full set of matrix statements, functions, and operators are provided to perform all standard computations on matrices.

VS BASIC provides a set of record-oriented input/output statements which allow the user to exercise detailed control over the transfer of information. Finally, VS BASIC can interface with external subroutines written in BASIC or other languages.

The VS is an interactive system whose primary means of program and data entry is the interactive workstation terminal. Each workstation consists of a large, easy-to-read CRT screen, used to display program messages, and a typewriter-like keyboard used to enter operator commands and responses.

The programmer accustomed to batch-oriented systems will find that an interactive system such as the VS offers many conveniences. Programs can be entered, edited, compiled, and run directly from the workstation keyboard, and results can be viewed at once on the CRT screen; the time-consuming intermediary steps of processing cards and printing out provisional results are completely eliminated. A complete set of system utilities also may be invoked directly from the workstation to perform common functions such as sorting, copying, and program linking. Finally, an extremely powerful interactive debugging facility is provided to enable run-time debugging of a program from the workstation. This debug facility also permits the programmer to examine and modify data values by referencing their symbolic data-names; it is not necessary to specify their absolute addresses in memory. All of these features are fully documented in the <u>2200VS Programmer's Introduction</u>.

For the program user, too, the interactive workstation is a distinct convenience. User instructions are displayed on the workstation screen in a clear, readable format. Data is entered from the keyboard and immediately validated; if correction is required, an appropriate message is displayed, and the data can be reentered correctly. The user can also interact with the BASIC program by selecting run-time options and supplying device and file information.

In summary, the interactive workstation provides a powerful and useful tool for the programmer and the program user alike. It enables the programmer to develop interactive programs which provide the user with far more control over program options and processing flow than was feasible in traditional, card-oriented systems. Because programs are entered, compiled, and debugged at the workstation, and because data validation and correction are performed automatically as data is entered, the tasks of program development and data entry are greatly facilitated on the VS.

1.2 HARDWARE CONFIGURATION

The Wang VS is an interactive, disk-based computer which supports multiple, concurrent tasks from interactive workstation terminals, and provides an efficient virtual memory addressing capability. The system is available with a variety of disk units, ranging from small, flexible diskette drives through medium-scale, fixed/removable disk drives, up to large-capacity, high-performance, multi-platter disk units. Industry-compatible, nine-track magnetic tape drives also are supported. For hardcopy output, a variety of printers, ranging from character matrix printers to high-speed line printers, are available. The primary device for programmer and program user communication with the VS system is the workstation, which is an interactive terminal consisting of a keyboard and a CRT display with 24 lines of 80 characters each.

1.3 ENTERING AND COMPILING A BASIC PROGRAM ON THE VS

The process of creating and running a BASIC program on the VS consists of three steps:

1. The BASIC source text is entered at the workstation using the VS source text EDITOR.

2. The source program is compiled into an object program using the VS BASIC compiler.

3. The object program is run.

These three steps can be accomplished by means of the Command Processor; this sequence is fully documented in Chapter 3 of the 2200VS Programmer's Introduction and summarized briefly in this section and the next. The entire sequence can also be accomplished by EZBASIC and the EDITOR, system utilities designed to provide integrated programming environments within which the processes of creating, editing, compiling, and running program files can be carried out easily and efficiently. EZBASIC and the EDITOR are fully documented in Chapter 6 of the 2200VS Programmer's Introduction and summarized below in the next few pages.

Logging On To The System

Before accessing any VS system resources, you must log on to the system by entering your User I.D. and password. When the LOGON procedure is successfully completed, the Command Processor Menu is displayed.

A program is run on the VS by invoking the RUN command (depress Program Function Key (PF) 1), and entering the program name as well as its library and volume. System utilities (such as the EDITOR, BASIC compiler, and EZBASIC) reside in the system program library, and do not require specification of a library or volume.

The VS contains two utilities which can be used to enter, edit, compile, and run BASIC programs - the EDITOR and EZBASIC.

The EDITOR

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

The EDITOR first displays an Input Definition screen. Type in B (or the full word BASIC) for language. If you are creating a new program, leave the input file name blank; the permanent source file is created and named later. If you wish to edit an existing program source file, enter the program name and the names of the library and volume to which it is assigned. If you are writing significantly more than 500 lines of new text, type an estimate of the number of lines in the space allotted. Otherwise leave that space blank. Now press ENTER.

The EDITOR next creates a work file for text editing. The editing of source text -- including entry of new text and modification of existing 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.

The EDITOR also now displays its normal menu, containing 14 functions for examining, entering, and editing source text. Program Function (PF) keys 1 to 7 are the Display Mode Keys. PF 8 to PF 14 are the Edit Mode Keys. The three most important functions are explained briefly here. More detail on these and explanations of the other functions may be found in the <u>2200VS Programmer's Introduction</u>.

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 in MOD mode, the line numbers supplied by the EDITOR can be altered in place, if the user wishes.

PF 12 - DEL (delete mode) allows the user to delete text from the source file, either a specific line, a range of lines, or ALL.

The normal menu also has two special functions: PF 15, which causes the column number of the current cursor position to be displayed (active in Display mode, modify mode, and insert mode) and PF 16, which causes the EDITOR Special Menu to be displayed. This menu is described below.

To enter lines of text, enter either MOD or INS mode and simply type in the lines. Pressing ENTER sends the lines which were just typed into the system for processing. This must be done after every line of INS mode. In MOD mode the screen may be filled up 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, you may press ENTER after typing in no new lines of text).

Now press PF 16 to reach the EDITOR Special Menu. The Special Menu has thirteen functions. The most important ones are listed below. These functions, as well as the others which are not described here at all, are described in detail in the <u>2200VS Programmer's Introduction</u>.

PF 1 - RETURN TO DISPLAY returns the EDITOR to the point in text editing from which the Special Menu was invoked.

PF 5 - CREATE creates a new file of the edited text. The user is asked to supply file, library, and volume names and several optional pieces of information, including a retention period during which only the file's creator and system security administrators can scratch or rename the file.

PF 6 - REPLACE replaces the old input file with the new edited text.

PF 9 - RUN compiles and runs an uncompiled program, or simply runs a compiled program. 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 the compilation is not necessary because it was done successfully in this EDITOR session since the last text entry, the program is run.

PF 10 - COMPILE invokes the BASIC compiler and the LINKER utility.

PF 11 - ERRORS appears only if the compilation errors were encountered after RUN or COMPILE were invoked. This key causes a list of the errors detected to be displayed. If the default value of ERRLIST in the Compiler/LINKER Options display was changed to NO, this key will not be displayed, and the error list will not be accessible from the EDITOR.

PF 16 - EOJ ends EDITOR processing and returns program control 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.

EZBASIC

The EZBASIC utility is an enhancement of the text EDITOR utility. EZBASIC can call the EDITOR directly, and thus perform all the functions of the EDITOR utility. It can also call the BASIC compiler and the LINKER, and can run system utilities or other programs without returning to the Command Processor.

To run EZBASIC, invoke the RUN command from the Command Processor, type EZBASIC in the space for PROGRAM, and press ENTER. The next screen is the default library and volume definition screen. This lists (1) a source library (by default the user's log on ID plus the letters SRCE) and volume, and (2) a program, or object, library (by default the user's logon ID plus the letters PROG) and volume. The default values may be changed. All source and object programs created in this EZBASIC session are automatically placed in these libraries, and when a program is run from the EZBASIC menu, the object library is used as the default location.

Once this screen is ENTERed, the EZBASIC menu is displayed. A space is allowed for a program name. The options are described briefly below. For more information see the <u>2200VS Programmer's Introduction</u>.

PF 1 - RUN runs the program whose name is inserted in the space allocated, if an object file for it exists either in the user's default library, defined on the first EZBASIC screen, or in the system library.

PF 2 - EDIT invokes the EDITOR.

PF 3 - COMPILE causes the source program with the program name given to be compiled, with linking by the LINKER utility.

PF 4 - UTILITIES displays a menu of system utility programs which can be invoked simply by pressing the appropriate PF keys.

PF 5 - MODIFY ENVIRONMENT allows the user to modify the Compiler/LINKER Options and the default source and object library and volumes.

PF 16 - END BASIC PROCESSING returns the user to the Command Processor.

The BASIC Compiler

BASIC source programs must be translated into machine language -- the internal instruction codes of the VS -- before they can be executed. This translation is carried out by a special program called a compiler. The compiler reads the BASIC source program, translates it into machine format, and stores the resulting program, called the <u>object program</u>, in an output file. The name of the BASIC compiler is, appropriately enough, BASIC.

Options

When BASIC is run, it first displays the list of compiler options, with the default value for each option. The complete list of options is documented in Appendix D. The three most important options are:

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

- SOURCE - This option 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. To suppress this listing, type NO.

- SYMB - This option directs the compiler to insert symbolic debug information into the object program. Symbolic debug information permits 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.

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 in the file name, along with its library and volume.

Output Definition

Unless 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 D), 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:

NUMBER OF RECORDS	The number of records in the output file is automatically determined by the compiler based on the size of the input file. This value should not, in general, be changed by the user.
RETENTION PERIOD	During the specified Retention Period, the file cannot be scratched or renamed. Only the owner can change the Retention Period. If such protection is not deemed necessary, the RETAIN field should be left blank.
RELEASE	If RELEASE=YES, any extra space in the object file is released for use by other files. Otherwise, it remains reserved for use by the object file.
FILE CLASS	The object file may be assigned to one of the VS file protection classes. Consult your system security administrator to determine in which protection class your file belongs.

When the output file name and all options have been defined, key ENTER.

Return Code

The program is now compiled by the BASIC compiler. When the compilation is completed, the Command Processor Menu is displayed, with a Return Code immediately above the menu. This Return Code indicates the status of the compilation:

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 is printed on the selected printer, followed by a list of compiler diagnostics. All other optional listings and tables are also printed.

The LINKER Utility

The VS LINKER can be used to perform the following functions:

1. Link two or more object program modules or subroutines into a single executable program.

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 optionally called whenever a program is compiled from either the EDITOR or EZBASIC. 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 2200VS Programmer's Introduction for more information on the LINKER.

1.4 RUNNING THE BASIC OBJECT PROGRAM

The compiled program is run with the RUN command from the Command Processor Menu. Depress 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.

If the program opens one or more data files, the operator must specify the file name(s), as well as the appropriate library and volume, during execution. (In the case of tape files, only the volume name and a file sequence number are necessary.) This information must be entered by the operator unless it is supplied by an accompanying procedure (procedures are discussed in the 2200VS Programmer's Introduction, Chapters 7 and 8) or by the program. (If the file, library, and volume names are supplied by the program, the GETPARM display requesting these will be displayed, with the names filled in, unless NODISPLAY or NOGETPARM were specified in the OPEN statement.)

1.5 FILE TYPES SUPPORTED ON THE 2200VS

The VS supports two file types, sequential and indexed. Sequential files contain records in strictly consecutive sequence, while indexed files are organized on the basis of designated key fields within the records. Both types of files may be accessed either sequentially or randomly, and a file may be accessed both sequentially and randomly within the same program.

1.6 PROGRAM DEVELOPMENT

BASIC programs are entered from a keyboard using the EDITOR utility (which can be reached directly from the Command Processor, or through EZBASIC). This program stores them on disk as "source" files. By using the BASIC compiler (which can be accessed directly, from the EDITOR, or from EZBASIC) to process the source file, an "object" file is produced which can be executed by the VS processor. The compiler performs a translation from the language specified in this manual to the machine language of the VS.

In addition to an object file, the compiler produces a printed listing. Optional components of this listing, which are selected when the compiler is invoked, include the following:

- A listing of the source file.
- A cross-reference listing of symbols and line numbers appearing in the program.

- A listing, in Assembly Language, of the machine language code which the compiler produced.

- Notification of any compilation errors detected by the compiler.

The programmer handles compilation errors and other errors detected after compilation by using the EDITOR to review and correct the source file, and then re-compile it.

The symbolic debugger is a powerful tool that the programmer uses to trace the execution of a BASIC program, and to detect the source of program errors which are not otherwise readily visible. The programmer must indicate to the compiler that it is to produce an object file which includes symbolic debugging capability. Such an object file, when executed, will provide an additional level of operator interaction, allowing the programmer to trace the progress of the program, and examine the values of its variables, referring to them by the names they were given in the source file. After identifying design or logic errors, the programmer typically edits the source file and re-compiles it to produce a correct object file.

1.7 SOURCE FILE FORMAT

A BASIC program consists of one or more 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:

100 RETURN	is a verb which forms a complete statement by itself. It signals the end of a subroutine.
100 LET X=2	LET assigns values to variables. In this case the variable X is assigned a value of 2.
100 GOTO 40	GOTO transfers control to the line number given, so processing is continued from there.
100 IF A=B THEN RETURN	among the additional information which may follow the IF...verb is another entire BASIC statement.
100 IF	IF by itself, however, is not a complete BASIC statement.

If a statement does not begin with a verb, it is presumed to be a "LET" statement.

X=2 — is a valid BASIC statement, and is equivalent to writing "LET X=2".

There are two types of BASIC statements: executable and nonexecutable. An executable statement specifies program action:

```
100 READ A,B
200 A = 6*B
```

A nonexecutable statement provides information for program execution:

```
100 DATA 1,4
```

or for the programmer:

```
200 REM THIS IS PROGRAM 1
```

but no specific action is taken when nonexecutable statements are encountered in the sequence of execution. The following BASIC statements are defined to be nonexecutable:

COM
DATA
DEFFN
DEFFN'
DIM
FMT
% (Image)
REM or *
SELECT, when used in File I/O (i.e., SELECT # and SELECT POOL)
SUB

The BASIC source file consists of lines. Lines in the source file contain 72 columns or character positions, which are numbered from left to right as columns 1 through 72. Each column may contain one character from the ASCII character set. There are no restrictions on the character set: uppercase and lowercase letters, numerals, and all symbols may be contained in a BASIC source file. However, lowercase letters can be used only as data in literal strings.

```
600 Let X=Sin(y) (lowercase is unacceptable.)
```

Columns 1 through 6 of each line contains a unique six-digit line number which identifies that line. (Note: for convenience, this manual drops leading zeros in line numbers.) The EDITOR utility takes care of line numbering for the user. If, in fact, the user supplies line numbers of his own in addition to the EDITOR supplied line numbers, they will result in error messages. Note that the user may edit line numbers in insert mode. Columns 7 through 72 may contain one or more statements in the BASIC language. Column 72 may contain an exclamation point (!) to indicate continuation, as described below. An asterisk (*) in column 7 causes the line to be ignored (i.e., treated as a REM statement) unless the previous line ended in a continuation character.

Spacing of the statements in columns 7-72 is up to the programmer. BASIC ignores blanks except within quotes. (Note: the words "space" and "blank" are used interchangeably in this manual.)

```
100 PRINT "A B C"
200 PRINT "ABC"
300 P R I N T "ABC"
```

Lines 100 and 200 will definitely have different effects. However, the added spaces in line 300 are superfluous; that line will have the same effect as line 200.

A line in the source file may contain more than one BASIC statement. A colon (:) is used to separate one statement from the next. For example:

```
400 LET X=5 : LET Y=5 : LET Z=5
```

Conversely, a BASIC statement may occupy more than one line in the source file. BASIC assumes that the end of a line marks the end of a statement in the same way that a colon does. However, an exclamation point in the rightmost column of a line (column 72) indicates that the rightmost statement in that line will continue on the next line. For example:

```
400 LET X                                               !
500 =4
```

Each line in the program must have a unique line number, even if it is used to continue a statement from a previous line.

Although a statement may begin on one line and end on another line, verbs, constants, and line number references may not be split between lines. For example:

```
400 LE                                                  !
500 T X=4
```

This is an example of an invalid way to split up a statement, since the verb "LET" is divided between lines. Literal strings, 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. Null lines (lines without a statement) are acceptable to the BASIC compiler, as is the use of the colon to form a null statement within a line. For example:

```
400 LET X=4 : LET Y=4 : : LET Z=4 :
500
```

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, FN, and, in certain cases, IF...THEN...ELSE.

1.8 BASIC LANGUAGE STRUCTURE

Programs written in BASIC must adhere to certain rules of syntax. These rules and the terms used to define them are described below.

Symbol Operators

The symbols used as operators in BASIC fall into three categories: arithmetic, relational, and assignment operators.

Arithmetic Operators

The following symbols are used as arithmetic operators.

Symbol	Sample Expression	Explanation
↑ or **	A↑B or A**B	Raise A to the power B.
*	A*B	Multiply A by B.
/	A/B	Divide A by B.
+	A+B	Add B to A.
-	A-B	Subtract B from A.

The following priorities are observed when evaluating expressions:

1. All operations within parentheses.

2. All exponentiation (↑ or **) is performed (from left to right).

3. All multiplication and division are performed (from left to right).

4. All addition, subtraction, and negation are performed (from left to right).

Quantities within parentheses are evaluated, using the above priorities, before the parenthesized quantity is used in further computations. 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. For example, in the expression A*B/C, A is multiplied by B and the result is divided by C.

Relational Operators

Relational operators are used in an IF...THEN statement when values are to be compared. For example, when the statement IF G<10 THEN 60 is executed, if 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 can be 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 in SEARCH.

Assignment Operator

The equal sign (=) is also used to indicate assignment of a value to a variable. For example, the statement A=10, when written in BASIC, indicates "assign the value 10 to the variable A". (Also see the LET statement.)

1.9 RULES OF SYNTAX

The following rules govern BASIC syntax.

1. Except in alphanumeric literals, % (Image) statements, and PIC clauses, BASIC ignores blanks. For example, the following statements are both valid and equivalent:

```
100 LET A = 2*B+C
100 LETA=2*B+C
```

2. Blank lines and null statements (: :) are allowed.

3. BASIC statements may be continued on as many lines as needed. The continuation character is "!", which must be present in the last column (column 72) if the line is to be continued. The continuation character may <u>not</u> break up a <u>keyword</u>, <u>constant</u>, or <u>line number reference</u>. A variable may be continued only if the initial part is sufficient to identify its type; for example, an alpha scalar may be continued after the "$", an array variable may be continued after the "(", etc.

   A literal may be continued for up to 255 characters. Note that all characters are considered part of the literal except the "!" in column 72.

4. Continuation lines have line numbers, just as do other lines.

5. Columns 1-6 contain the line number, so only 66 columns are actually available for program text.

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,

```
              {literal       }
   operand =  {alpha variable}
              {expression    }
```

   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 as needed can be added to the INPUT statement.

6. The order of parameters shown in the general form must be followed.

CHAPTER 2 NUMERICS

2.1 INTRODUCTION

VS BASIC supports two classes of numeric data: floating-point and integer. The classes 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 are used to represent "whole" (i.e., non-fractional) numbers, are stored in four bytes of memory. Integer operations are considerably faster than floating-point operations. Floating-point data are stored in eight bytes of memory in the form of: (1) a hexadecimal fraction between 0 and 1, and (2) a power of 16. This format provides a convenient way of representing numbers whose magnitude is either extremely large or extremely small, 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 classes of data in arithmetic expressions and assignment statements. Resulting expressions, which are called mixed mode, are covered at the end of this chapter.

A third data class, alphanumeric, which also has distinct representation in syntax and internal format, is used to represent literal data. Alphanumeric data is discussed in Chapter 3.

Each class of data is further broken down into two subclasses: constants and variables. The value of a constant is fixed and does not change during program execution. Variables do not have fixed values and can be assigned new values during program execution. A numeric constant is represented in a BASIC source file by a number. A numeric variable, on the other hand, is represented by a symbol (the "variable name"). This symbol is

used to name that area in storage which holds the value of the variable. In the following statement:

```
100 X = 3.25
```

X is a variable name, 3.25 a constant. The statement itself, an <u>assignment statement</u>, assigns the value 3.25 to the variable X.

2.2 NUMERIC CONSTANTS

Floating-point

A floating-point constant may be a positive or negative number of up to 15 digits. Numbers with more than 15 digits result in a warning, and only the first 15 digits, excluding leading zeros, are used by BASIC statements or functions.

The magnitude of a floating-point constant can be 0 or range from 16^{-65} to 16^{63} (approximately 5.4×10^{-79} to 7.2×10^{75}).

Very large or very small numbers can be expressed in exponential form.

For example, 4.5×10^{7} is written as 4.5E+07 and 4.5×10^{7} is expressed as 4.5 E-07. Exponents must be integers. Leading 0's in the exponent may be omitted; if the sign is omitted, + is assumed.

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

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

The following show invalid use of scientific notation:

8.7E5.8	Not valid because of the illegal decimal form of the exponent.
.87E-99	Not valid because it is less than 5.4E-79.
-103.2E99	Not valid because it is greater than 7.2E75.

NOTE:

In general, decimal numbers may contain any number of digits provided that the value of the number itself is within legal bounds. Any BASIC statement or function dealing with such numbers will use only the 15 most significant digits.

This applies in the following situations:

1. Any number in the source file.

2. Any user-input value (e.g., INPUT, ACCEPT).

3. Any number converted via Image (%) or FMT (PIC) (e.g., ACCEPT DISPLAY, PRINTUSING, disk I/O statements).

4. Any internal character-string representation of such a number (e.g., CONVERT, NUM).

<u>Integer</u>

An integer constant may range from -2,147,483,648 to 2,147,483,647 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 floating-point. However, the percent sign for numeric constants is only permitted for numbers actually contained in the source file. Therefore, numbers given to the program during execution (i.e., from the workstation or from a data file or converted from an alpha-expression) are given in floating-point form, without the percent sign.

2.3 NUMERIC VARIABLES

Numeric variables are used to store numeric data in memory. Unlike constants, whose values are fixed and cannot be changed during program execution, 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 which is used to contain the value of that variable. Numeric variables are initialized to zero by the compiler.

Within each class of numeric variable, there are two distinct data types: scalar variables and array variables, which differ both in how they are handled in the syntax and in how storage is allocated for them. A numeric scalar variable can contain a single numeric value. All floating-point scalar variables are eight bytes in length, while all integer scalar variables are four bytes in length.

A floating-point scalar variable is designated by a single uppercase letter (A-Z) or a letter followed by a digit (0-9). There are thus 286 floating-point scalar variable names. Integer variable names are distinguished from floating-point variable names by the presence of a percent sign ('%') immediately following the variable name. For example, N is a floating-point variable name, while N% is an integer variable name. Similarly, A3 is a floating-point variable name, A3% an integer variable name. Examples of valid numeric variable names are:

Floating-point	Integer
A	A%
A1	A1%
B0	P9%
N8	B1%

A floating-point variable name and an integer variable name always identify different variables, even if the names (i.e., the letters, or the letters and digits), are identical. For example, N2 and N2% identify two different variables, one floating-point and one integer, and both may be used to refer to different pieces of data in the same program without ambiguity.

<u>Array Variables</u>

An "array variable" is really a collection of scalar variables identified by a common name. The array as a whole is referred to by the array name, plus two parentheses to form an "array-designator" (e.g., B()). The array name alone (e.g., B) is used only in special matrix statements (e.g., MAT INPUT and MATPRINT). Each of the scalar variables contained in the array is referred to as 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. A subscript is denoted by a pair of parentheses enclosing the number. For example, the fifth element in array N() could be specified as N(5). Note that the subscript is enclosed in parentheses immediately following the array name. The names of array variables are formed in exactly the same way as the names of scalar variables (that is, a letter or a letter and a digit).

Since scalar variables are different from array variables, the same name (i.e., the same letter, or the same letter and digit) 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.

In general, any reference to an array variable must consist of the array name immediately 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. In the example cited above, N(5) identified the fifth element of array N(). If the syntax allows the entire array, rather than a particular element of the array, to be referenced, the array name must be followed by empty parentheses, e.g., N() or A%(), to form the array-designator. An array can have the same name as a scalar variable, but the array must (with few exceptions) always be referenced with an array-designator, to indicate that an array, rather than a scalar variable, is meant. For example:

N6	identifies a floating-point scalar variable.
N6%	identifies an integer scalar variable.
N6()	identifies a floating-point array.
N6%()	identifies an integer array.

One-Dimensional Arrays and Two-Dimensional Arrays

Array variables are of two types, one-dimensional and two-dimensional. A one-dimensional array is a "list" of variables all identified by the same name. A two-dimensional array is a "table" of variables, all identified by the same name.

One-dimensional arrays are also called "lists," "vectors," "column vectors," and, since each element is identified by a single subscript, "singly-subscripted arrays." In general, the term preferred by a programmer is the one which makes the most intuitive sense for his application: programmers involved in data processing tend to prefer "list," while those programming mathematical applications tend to be more comfortable with "vector."

A one-dimensional array can be conceived 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 N() in Figure 2-1.

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

Figure 2-1. The One-Dimensional Array N()

Note that N() contains a total of five elements and that each element is identified by specifying its row in the column (e.g., element N(3) is located in row 3).

It is not difficult to generalize the scheme in Figure 2-1 to contain two or more columns. When this is done, the result is a two-dimensional array. Two-dimensional arrays are also called "tables," or "matrices," and because each element is identified by a pair of subscripts, "doubly-subscripted arrays."

A two-dimensional array can be conceived as a table consisting of two or more columns of elements. Consider, for example, the representation of the two-dimensional array M() in Figure 2-2.

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

Figure 2-2. The Two-Dimensional Array M()

Note that M() consists of two columns of elements, with five rows in each column, for a total of 10 elements. In this case, it is not sufficient to identify each element by its row, since the element may be in column 1 or column 2. 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 M(3,2) identifies the element in the third row of column 2.

Elements in an array can be referred to by subscripts that are legal BASIC expressions. Thus J(N) refers to the Nth element of array J() 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 the array, since it can eliminate a considerable amount of repetitive coding. For example, the following three statements

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

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

<u>Dimensioning an Array</u>

Although arrays in BASIC default to 10 by 10, they can be explicitly dimensioned to from 1 to 32,767 elements per dimension. If a value other than the default is to be used, the program must specify the size of the array before any other references are made to the array. This is done by means of the <u>dimension</u> (DIM) or <u>common</u> (COM) statement. (The COM statement also reserves a common area of memory for variables that are used in several programs - a <u>calling program</u> and one or more <u>subroutines</u>. This enables variables whose values are changed by one program to be passed with these changes to another variable.)

> NOTE:
>
> The total size of all the variable in a program, including array variables, is limited to a maximum of not more than 512K bytes. The exact maximum will depend on the particular circumstances under which a program is run.

The DIM statement (or the COM statement) defines the amount of memory allocated to an array prior to program execution. The following DIM statement defines a one-dimensional integer array of 10 elements:

```
100 DIM A%(10)
```

A DIM statement can 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. The following DIM statement defines a one-dimensional integer array, a two-dimensional integer array, a one-dimensional floating-point array, and a two-dimensional floating-point array:

```
100 DIM B%(5), C1%(2,5), A(10), C1(10,10)
```

If an array is not dimensioned by a DIM or COM statement, VS BASIC gives it default dimensions of 10 by 10.

Note that since DIM statements are processed during compilation, prior to program execution, they _cannot_ 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:

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

2.4 EXPRESSIONS

A numeric expression may consist of a single variable or constant, or it may consist of a series of variables and constants connected by arithmetic operators and numeric functions. An expression may be preceded by plus or minus and may be contained within parentheses. In the following examples valid BASIC expressions are boxed:

```
X      =    |A|

X      =    |5*Y+FNB(X) - LOG(I)|

F( |X2+5| ,|K| |9|

FOR I = |3+K2| TO |4*Y| STEP |D(3+K)-1 |

PRINT   |SIN(K)-4*J|
```

Numeric expressions can be evaluated in a variety of different BASIC statements. Most commonly, expressions are evaluated and their values assigned to variables in assignment (LET) statements, or they are evaluated and their values printed or displayed in PRINT statements. Operations in an expression are executed in sequence from highest priority level to lowest (see Section 1.11).

2.5 MIXED MODE

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

1. [LET] assignment statements allow mixed-mode assignment.

2. Statements performing implicit assignment, such as CONVERT, GOSUB'(), FN(), INPUT, ACCEPT, and READ, allow mixed-mode. The only exception to this is the CALL/SUB procedure, which does not allow mixed-mode argument passing.

3. The percent sign (%), used to indicate an integer value, can 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
   ```

2.6 NUMERIC FUNCTIONS

A function is an item in the BASIC language which returns to the program an output value based on input values. As input to the function, the programmer provides one or more expressions (called arguments of the function). As output, the function returns a value which is made available for use in a larger expression. The value returned by the function will depend on the input values.

Syntactically, a function is written as follows:

```
function name[(argument[,argument][,...])]
```

where "function name" is the name of one of the system-defined functions, and "argument" is any expression which is 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, in the following order: (1) the expression Y/2 to be calculated, (2) the sine of that expression to be determined, (3) 1 to be added to the sine, and (4) the assignment of the final value to the variable 'X'.

BASIC provides all the standard trigonometric, logarithmic, and exponential functions. In addition, greatest-integer, signum (see description of SGN, below), and absolute-value functions are provided, as are a random-number generator and several functions which accept alpha strings as arguments and return numeric values.

The BASIC programmer can define customized functions using the DEFFN statement. For instance, having included the statement

```
100 DEFFNA(X)=(EXP(X)-EXP(-X))/2
```

the programmer can use the user-defined function FNA(numeric expression) anywhere in a numeric expression to return the hyperbolic sine of the expression inside the parentheses.

The SIZE function, which deals with file I/O, is discussed in Chapter 6.

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.

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)	same (ATN is a synonym for ARCTAN).

These functions can express angular measure in one of three modes: (1) degrees, (2) radians, or (3) 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 D which selects degrees.
SELECT G which selects grads.
SELECT R which re-selects radians.

The mode used at any time is determined by the most recently executed SELECT (D,G, or R) statement in that program or subroutine. For instance, a program can execute a "SELECT D" 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 G" 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 II. 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 interpreted as degrees, grads, or radians according to the SELECT setting.

Other Numerical Functions

The remaining nineteen 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.
DIM (see below)	DIM(X(),1)	The maximum 1st or 2nd subscript of the array X.
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.

Function Name	Sample Expression	Meaning
INT	INT(X)	The greatest-integer function; the greatest integer less than or equal to the argument.
LEN	LEN(A$)	The actual length, in bytes, of the argument.
LGT	LGT(X)	Common (base 10) logarithm.
LOG	LOG(X)	Natural (base "e") logarithm; inverse function of EXP.
MAX	MAX(X,Y,Z)	The value of the largest element in the argument list.
MIN	MIN(X,Y,Z)	The value of the smallest element in the argument list.
MOD	MOD(X,Y)	The modulus function; the remainder of the division of the first element by the second.
NUM	NUM(A$)	The number of sequential ASCII characters in the argument that represent a legal BASIC number.
#PI	#PI	The value 3.14159265359.
POS	POS(A$<B$)	The position of the first character of the first argument which is < , <=, >, >= , <>, or = the first character of the second argument.
RND (see below)	RND(X)	A pseudorandom number between zero and one.
ROUND (see below)	ROUND(X,N)	The value of the first argument, rounded off to the accuracy specified by the second argument.

Function Name	Sample Expression	Meaning
SGN	SGN(X)	The signum function; -1 if the argument is negative, 0 if the argument is zero, or +1 if the argument is positive.
SIZE	SIZE(#1)	The size in bytes of the most recently read record from the specified file.
VAL	VAL(A$,3)	The numeric value of the first 1, 2, 3, or 4 bytes of the first argument.

The RND, ROUND, and DIM functions must be discussed in more detail.

RND

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 employs 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 random; thus the term "pseudorandom."

There are three ways to use RND(exp), based on the value of the argument:

1. exp1 < 0 or exp1 ≥ 1

   If the argument is less than zero or greater than or equal to one, RND produces the next pseudorandom number in the "list," as described above. If this is the first use of RND in the program, the "previous" value is assumed to be some arbitrary value set by the BASIC compiler at compilation; thus, the same value will be produced until the program is re-compiled.

2. 0 < exp2 < 1

   If the argument is between zero and one, RND returns the argument itself as the result and resets the "list" to this value. The next use of RND as in Option 1 above (expl < 0 or > 1) will use the value of the argument (exp2) as the "previous" 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 "list" of random numbers to a random value, so that on subsequent calls using Option 1, a 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 will not produce a dependably random list of numbers. To produce such a 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.

Examples:

```
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↑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 (1-N)th place 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   etc.
```

Note that ROUND, unlike INT, will round up as well as down; for example, if ROUND is told to round 4.7 to 0 decimal places, it will produce 5, not 4.

DIM

The DIM function (not to be confused with 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 a digit 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.

Just as DIM may be used to determine the dimensions of an array, an expression of the form LEN(STR(alpha-expression)) may be used to determine the defined length of an alpha variable or array string, as discussed in the following section.

2.7 SUMMARY OF RULES, FORMATS, AND SYNTAX

Floating-Point

The following are classified as floating-point values:

1. Any floating-point variable (no '%').
   E.g., A, B1, C(3), D4(X,5)

2. Any numeric constant with no '%'.
   E.g., 5, 3.7, -6E3, 1E-1

3. The result of any valid numeric function except FNa%, LEN, NUM, POS, VAL, SGN, SIZE, DIM, ABS(integer), and under certain conditions MIN, MAX, and MOD.
   E.g., FN2(2%), SIN(3), ABS(-12)

4. The result of any binary operation (+, -, *, /, ↑) or MOD function whose two arguments are not both integers.
   E.g., 2/5, 3%↑17, 4%+SQR(16)

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

NOTE:
Any intermediate or final floating-point overflow causes an error; underflow becomes 0 with no error if encountered during program execution. A numeric constant which is either too small or too large generates a compile-time error.

Integer

The following are classified as integer values.

1. Any integer variable (with '%').
   E.g., A%, B1%, C%(3), D4%(X,5)

2. Any integer constant, which must contain a trailing '%', no decimal point, and no exponent.
   E.g., 375%, -10000%, 2%

3. The result of the numeric functions FNa%, LEN, NUM, POS, VAL, SGN, SIZE, DIM and ABS (integer).
   E.g., FN3%(7.5), SGN(X), SIZE (#5)

4. The result of any binary operation (+, -, *, /, ↑) or MOD function whose two arguments are both integers.
   E.g., 2%/5%, 3%↑(-17%), 4% + LEN(B$)

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

NOTE:
Any intermediate or final integer value lying outside the allowable range causes an error.

Numeric Terms

```
                        [     ]
1.    constant:         |{+}|     {floating-point constant}
                        |{-}|     {integer constant}
                        [     ]

2.    expression ≠:        {numeric variable                                      }
      (or exp)             {constant                                              }
                           {mathematical function                                 }
                           {DIM function                                          }
                           {FN function                                           }
                           {LEN function                                          }
                           {NUM function                                          }
                           {POS function                                          }
                           {SIZE function                                         }
                           {VAL function                                          }
                           {               [{+}               ] [    ] }
                           {               |{-}               | |    | }
                           {[{±}]expression|{*} expression    | |... | }
                           {               |{/}               | |    | }
                           {               |{↑}               | |    | }
                           {               [{**}              ] [    ] }
                           { (expression)                                         }
```

3. *int*: digit [digit]...[%]

4. *numeric array-designator*: letter [digit] [%] ()

5. *numeric array name*: letter [digit] [%]

6. *numeric array variable*: letter [digit] [%] (exp [,exp])

7. *numeric scalar variable*: letter [digit] [%]

8. *numeric variable*: {numeric scalar variable}
   {numeric array variable }

9. *mathematical function*: #PI, MAX, MIN, MOD, ROUND, ABS, ARCOS, ARSIN, ARCTAN, ATN, COST, EXP, INT, LGT, LOG, RND, SGN, SIN, SQR, TAN functions.

≠ adjacent operators are not allowed (e.g., A++B).

CHAPTER 3 ALPHANUMERICS

3.1 ALPHANUMERIC CHARACTER STRINGS

In addition to its ability to manipulate and operate upon numeric values, VS BASIC also provides an extensive 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 combination of keyboard characters, including letters A-Z, numbers 0-9, and special symbols +, -, $, etc. Characters not found on the keyboard can be represented in the form of hexadecimal codes. Typical examples of uses of character strings are names, addresses and report headings.

Character strings are represented in a program in two basic forms:

1. As the values of alphanumeric string variables, or portions of string variables.

2. As literal strings, the alphanumeric equivalents of numeric constants.

Both alphanumeric scalar variables and alphanumeric array variables may be used. The dimensions of alpha arrays can be specified in a DIM or COM statement prior to their use in the program, or the defaults (variable length of 16 bytes, array dimensions 10 by 10) can be used.

3.2 ALPHANUMERIC STRING VARIABLES

Alphanumeric character strings are stored and processed in a special type of variable called the alphanumeric string variable, or simply alpha variable. Alpha variables are distinguished from numeric variables by the presence of a dollar sign ('$') following the variable name. For example, the variable name 'A' represents a floating-point variable, while the variable name 'A$' represents an alpha variable; similarly, 'N3%' is integer, while 'N3$' is alphanumeric, etc. Note that a numeric variable and an alphanumeric variable are separate and independent variables. Data stored in an alpha variable cannot be operated on by arithmetic operators, i.e., +, -, *, /, ↑ or **. There are, however, a number of logical operators (discussed in Section 3.8 in this chapter) which can be used to test and manipulate alphanumeric data.

Alpha variables are of two types: alpha scalar variables and alpha array variables. An alpha scalar variable can store a single character string from 1 to 256 characters in length. An alpha array variable consists of one or more array elements, each of which can store a character string from 1 to 256 characters in length. Array variables are useful because they enable the programmer to reference a collection of data with a single array name. (The separate character strings stored in the elements of an alpha array can be treated together as a single contiguous character string using the STR function. See STR function in Part II.)

Alphanumeric array variables, like numeric arrays, are further divided into two classes: one-dimensional arrays and two-dimensional arrays. One-dimensional arrays (also called "single-subscripted arrays") may be conceived as analogous to lists, with a single row of elements. Two-dimensional arrays (also called "double-subscripted arrays") are analogous to tables, with both rows and columns of elements. The structure of arrays is discussed in greater detail in Chapter 2, Section 2.3.

Scalar variables and array variables are regarded by the system as separate and independent types of variables, while one-dimensional and two-dimensional array variables are different but related kinds of arrays. Thus the same name may be used in a program for both an alpha scalar variable and an alpha array variable, but the same name cannot be used for a one-dimensional alpha array and a two-dimensional alpha array in the same program. For example, the variable names A$ and A$(5) can both be used in the same program, but A$(5) and A$(6,6) cannot.

The DIM and COM statement can be used to define the amount of memory allocated to a scalar or array variable; in a DIM or COM statement, the expressions following the array name contain constants giving the row and column dimensions of the array. The DIM or COM statement must precede the first reference to the array or any of its elements in the program, or the default size, 10 by 10, will be used.

Examples of DIM and COM statements:

```
DIM A$(10), C3$(1,5), F7$(2,4)
COM A$(12), C5$(2,2), F1$(1,2)
```

> **NOTE:**
>
> An alpha scalar or alpha array element is filled with blanks (HEX(20)) when the scalar or array is initially defined.

Alphanumeric Variable Length

An alphanumeric variable identifies a unique location in memory reserved for the storage of alphanumeric data. The system reserves space for each variable during compilation, at which time the program is scanned for all variable references. The amount of space reserved for each variable can be specified by the programmer in a DIM or COM statement. The maximum length of an alpha scalar variable or of an element in an alpha array is 256 bytes, while the minimum length is one byte in each case. If the programmer does not explicitly dimension an alpha variable in a DIM or COM statement, the system automatically reserves 16 bytes for the variable. Similarly, if the programmer does not specify an element length when dimensioning an alphanumeric array, the system automatically reserves 16 bytes for each element of the array.

> **NOTE:**
>
> If a value other than the default (16 bytes) is to be used for a variable, the DIM or COM statement which dimensions that variable must appear before any other reference to the variable.

The length of an alpha variable or 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 end of the value of an alpha variable is normally assumed to be the last nonblank character (except when the value is all blanks, in which case the value is assumed to be 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 "<u>current</u> 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$="ABCD    "
200 PRINT LEN(A$)
4
```

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

Most alphanumeric instructions operate on the current length of an alpha variable. (That is, they operate on the current value up to the first trailing blank.) In some cases, however, the entire defined length of the variable may be used. It is important, therefore, to understand the distinction between defined length and current length.

> NOTE:
>
> If the defined length of an alpha scalar or array element is larger than needed for storing a given value, the scalar or array element is filled out with trailing blanks (HEX(20)) when the value is stored.

3.3 LITERALS

Alphanumeric Literal Strings

An alphanumeric literal string is a character string enclosed in double quotation marks ("). Literal strings can be specified as constant data, usually in a PRINT statement, to create headings or titles. For example:

```
100 PRINT "VALUE OF X=";X
```

In line 100, the character string VALUE OF X= is a literal string which is printed exactly as it appears.

Literal strings can also be assigned to alphanumeric variables. For example:

```
100 A$="BOSTON,MASS."
200 PRINT A$
```

Output: BOSTON,MASS.

A Literal string may be from 1 up to 255 characters in length. However, when a literal string is stored in an alpha variable, it is truncated to the defined length of the alpha variable. For example:

```
100 DIM A$5
200 A$="123456789"
300 PRINT A$
```

Output: 12345 (Note that the value was truncated to five characters since the defined length of A$ is five bytes.)

The minimum length of a literal is one; the null string ("") is not allowed. The double-quote character is not allowed within a (double-quote) literal string.

Alphanumeric literals can in general be used wherever alpha variables are allowed, except where the statement requires a receiver and does not permit the use of an alphanumeric constant. (See definitions of alpha-receiver/expression.)

Lowercase Literal Strings

A second type of 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 converted to lowercase by the system.

For example,

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

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

> NOTE:
>
> Lowercase strings can be indicated using single-quotes <u>without</u> setting the EDITOR to UPLOW (which allows lowercase letters to be entered from the keyboard directly). Strings enclosed in single-quotes are entered as uppercase; they are <u>output</u> as lowercase.

Any character is valid in a lowercase literal string <u>except</u> the single-quote character ('). Note that a single-quote (lowercase) literal string may contain double quotes, and vice-versa.

<u>Examples of Statements Using Alpha Variables And Literal Strings</u>

Alphanumeric string variables can be used in the BASIC statements listed below. Literal strings can generally be used in place of alpha variables, except where a value is assigned to the variable.

```
LET          LET A$=B$(2)
             A$="ABCD"
IF...THEN    IF A$=B$ THEN 100
             IF A$<"DR" THEN 200
             IF "ABCD">B$ THEN 300
INPUT        INPUT A$,B$(4)
READ         READ C$,D$,E$(1,2)
PRINT        PRINT A$,B$,"ABCD"
PRINTUSING   PRINTUSING 50,A$,B$,"LAST"
DATA         DATA "ABCD","EFGH",10
STR          A$=STR(B$(I),I)
```

3.4 ALPHA-RECEIVERS AND ALPHA-EXPRESSIONS

There are two main types of alphanumeric arguments used with alpha functions and statements in BASIC: alpha-receivers and alpha-expressions.

Alpha-Receivers

An alpha-receiver is an alphanumeric item which has a specific memory location, such as a variable or array. Alpha-receivers must be 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 6.6)

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

Alpha-Expressions

Alpha-receivers are a special case of the more general <u>alpha-expression</u>, which is a combination of receivers, literals, alphanumeric functions, the concatenation operator (&) and optional parentheses. Alpha-expressions may be used wherever variable values are allowed as "sending", as opposed to "receiving", fields, e.g., on the right side of a LET statement, in the arg list of a WRITE statement, etc. The maximum length of an alpha-expression is 32,767 characters.

The following is a general list of allowed alpha-expressions:

alpha-receiver	(e.g., A$, STR(A,$1,1))
literal	(e.g., "A", HEX(00))
alpha exp & alpha exp	(e.g., A$ & "XX")
(alpha exp)	(e.g., (A$))
BIN function	
DATE function	(see description in Section 3.9)
TIME function	
FS function	(see Section 6.6)
MASK function	(see Section 6.6)
STR function	

Note that all alpha-receivers are alpha-expressions, but the converse is not true.

e.g. <u>Alpha-expressions</u>

Receiver	Non-receiver
A$	(A$)
STR(X$,1)	"LITERAL"
KEY (#1)	A$ & "xx"
X1$()	DATE
STR (A$())	BIN (X↑2,3)

3.5 CONCATENATION OF STRINGS

The concatenation operator (&) combines two strings; one string is put directly after another, 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 alphanumeric operand, 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"
300 PRINT C$
```

Output: APRIL IS THE CRUELEST MONTH, BREEDING

3.6 ALPHA ARRAY STRINGS

An entire alpha array can be treated as a single (long) alpha variable when an alpha variable would be allowed. In this case the alpha is referred to by its name followed by "()" (the same syntax used for alpha array-designators). The array is treated as a single contiguous character 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$(2,2)="4"
300  PRINT A$()
```

Result: 1 2 3 4
(Note that *final* trailing spaces are not printed, exactly like ordinary alpha variables.)

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 a designated array.

3.7 HEXADECIMAL LITERAL STRINGS

Hexadecimal literal strings are a special form of literal string consisting of one or more hexadecimal codes specified in a HEX function. (See the discussion of the HEX function.) Hexadecimal codes are composed of a pair of hexadecimal digits (0-9 or A-F); they are particularly useful for representing ASCII characters not found on the workstation keyboard and workstation control codes (Field Attribute Characters). For example, hexadecimal codes can be used to format the CRT display into fields (see the discussion of field attribute characters in Chapter 5, Section 5.5).

HEX literal strings are legal wherever alphanumeric literal strings are allowed. In particular, they can be assigned to alpha variables in an assignment statement. For example:

```
600 A$ = HEX(313233)
700 PRINT A$
```

Output: 123 (The characters "123" are printed, since they are represented by the hex codes 31, 32, and 33.)

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 to FF - field attribute characters.

3.8 LOGICAL EXPRESSIONS

There is a special type of alpha-expression, allowed only on the right side of a LET (assignment) statement, which uses logical operators, such as AND, OR, etc. These expressions are called logical expressions and are defined below.

```
Logical expressions  (LET statement only)

General form:

 [operator] operand [operator operand] [operator operand][...]

 where operator =    ADD[C]
                     AND
                     OR
                     XOR
                     BOOLh

         operand =   alpha-expression
                     ALL function
```

Note that the operand list is virtually the same as that for alpha-expressions, with the addition of the ALL function, defined in the next section. Also note that concatenation (&) and parentheses are not allowed within logical expressions.

Evaluation of Logical Expressions

A statement of the form: LET 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. In all cases, the defined lengths of both arguments are used, with the operation proceeding one byte at a time as follows:

   - AND, OR, XOR, BOOLh

     The operation proceeds 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.

- ADD[C]

  The operation proceeds 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.

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

Note that 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$.

Logical expression operators

(For examples, assume DIM A$2)

1. ADD
   Adds the binary values of the arguments, one byte at a time with no carry propagation.
   ```
   E.g., 100  A$ = HEX(0123) ADD HEX(00FF)
    Result:   A$ = HEX(0122)
   ```

2. ADDC
   Adds the binary values of the arguments, one byte at a time with carry propagation (like 2 long binary numbers).
   ```
   E.g., 100  A$ = HEX(0123) ADDC HEX(00FF)
    Result:   A$ = HEX(0222)
   ```

3. AND
   Logically AND's the two arguments, one byte at a time.
   ```
   E.g., 100  A$ = HEX(0FF0) AND HEX(0F0F)
    Result:   A$ = HEX(0F00)
   ```

4. OR
   Like AND, but logically OR's the two arguments.
   ```
   E.g., 100  A$ = HEX(0F0F) OR HEX(0FF0)
    Result:   A$ = HEX(0FFF)
   ```

5. XOR
   Like AND, but logically exclusive-OR's the arguments.
   ```
   E.g., 100  A$ = HEX (0F0F)X OR HEX (0FF0)
    Result:   A$ = HEX(00FF)
   ```

6. BOOLh

A generalized logical operator, whose function is determined by the digit "h".

```
E.g., 100  A$ = HEX(0FF0) BOOL8 (HEX(0F0F)
 Result:   A$ = HEX(0F00)
```

3.9 FUNCTIONS WITH ALPHA ARGUMENTS

Four functions enable a BASIC program to evaluate the contents of an alpha-expression in predefined ways. All four return an integer value.

Function Name	Sample Expression	Meaning
LEN	LEN(X$)	The length of the argument.
NUM	NUM(X$)	The number of consecutive characters, starting at the first, from the argument which forms a legal BASIC (ASCII) number.
POS	POS(X$="$")	The position within the argument of a specified character.
VAL	VAL(X$,N)	The binary numeric value of the first N characters of X$.

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.

LEN("ABCDE")	Returns 5%.
LEN(E$)	Returns the actual length of E$.
LEN(STR(E$))	Returns the defined length of E$.
LEN(A$&B$)=LEN(A$)+LEN(B$)	This relation is always true.

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, 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, however, the argument is entirely blank, NUM returns 0%.

NUM can be used to validate an alphanumeric representation of a number before attempting to convert it to internal numeric binary form.

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.

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 with respect 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 hex 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 string variables (e.g., IF A$ < "ABCD"), 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 variables 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, so long as all their other characters are equal.

Example:

```
100 DIM A$4,B$5,C$5                 800  GOTO 1000
200 A$="ABC"                        900  PRINT "A$=C$";A$,C$
300 B$=HEX(41424321)                1000 PRINT HEXOF (A$)
400 C$="ABC  "                      A$=C$ABC      ABC
500 IF A$=B$ THEN 700               41424320
600 IF A$=C$ THEN 900
700 PRINT "A$=B$";A$,B$
```

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 1, 2, 3, or 4 characters from the alpha-expression, depending on the value of the second argument, and returns an integer value comprised of the binary value of the extracted character(s).

VAL(A$) or VAL(A$,1) will simply return the binary 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 between 0% and 255%, inclusive. Note that the binary value of characters is as specified by the ASCII code.

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% to 65535%, inclusive.

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 statement can be used to extract characters from an integer expression which contains their binary values, reversing the operation performed by VAL.

Alphanumeric Functions

Eight BASIC functions return alphanumeric values.

Function Name	Sample Expression	Meaning
ALL	ALL(A$)	defines a character string consisting entirely of characters equal to the first character of the alpha-expression.
BIN	A$=BIN(A,d)	converts the integer value of an expression to an alphanumeric string of d characters (1, 2, 3, or 4) which is the binary equivalent of the expression.
DATE	DATE	returns a six-character string giving the current date in the form YYMMDD.
FS	FS(#1)	returns a two-character code representing the file status for the most recent I/O operation involving the specified file.

KEY	KEY (#1)	returns the "key" field for the last record read from the specified file.
MASK	MASK (#1)	returns the two-character alternate key mask for the last record read from the specified file.
STR	STR (A$,A,B)	specifies the substring of an alpha variable or array string.
TIME	TIME	returns an eight-character string giving time to the hundreth of a second in form HHMMSShh.

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

3.10 SUMMARY OF RULES, FORMATS, AND SYNTAX

Alphanumeric Length

1. [ACTUAL] LENGTH (in bytes)
   (as determined by LEN function, also called "current" length)

   - alpha variable
     Does not include trailing blanks.
     If all blank, length=1.

   - alpha array string
     Like a single long alpha variable.

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

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

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

   - Literal
     Length is the number of characters within quotes or the number of hexdigit pairs in HEX.

   - FS function
     Length=2.

- DATE function
  Length=6.

- TIME function
  Length=8.

- MASK function
  Length=2.

- BIN function
  Length as specified in BIN (1,2,3, or 4; default=1).

2. DEFINED LENGTH

- alpha variable
  As specified in DIM, COM, or most recent MAT REDIM. (Default = 16.)

- 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.)

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

- All other alpha forms
  Same as actual length.

> NOTE:
>
> Receiving fields and fields treated as data buffers (e.g., PUT) always have their defined lengths available to the operation.

Alphanumeric Terms

1. alpha scalar variable: letter [digit]$

2. alpha array name: letter [digit]$

3. alpha array-designator: letter [digit]$()

4. alpha array element: letter [digit]$(exp[,exp])

5. alpha variable: {alpha scalar variable}
   {alpha array variable }

6. alpha array string: letter [digit]$()
(Treated as a single long alpha variable)

7. literal:

```
{                 [              ]     }
{"{any       }  |{any        }|    "}
{ {character}  |{character}|... }
{ {except    }  |{except     }|     }
{ {  "       }  |{  "        }|     }
{                 [              ]     }
{                                      }
{                 [              ]     }
{'{any       }  |{any        }|    '}
{ {character}  |{character}|... }
{ {except    }  |{except     }|     }
{ {  '       }  |{  '        }|     }
{                 [              ]     }
{ HEX(hh[hh]...)                       }
```

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

9. alpha-receiver:

```
{alpha-variable                           }
{STR(alpha receiver[,[exp][,exp]])        }
{alpha array string                       }
{KEY (file-expression [,exp])             }
```

10. alpha-expression: (or alpha-exp)

```
{alpha-receiver                           }
{literal                                  }
{DATE function                            }
{TIME function                            }
{BIN function                             }
{MASK function                            }
{FS (file-expression)                     }
{alpha-exp & alpha-exp                    }
{(alpha-expression)                       }
{STR (alpha-exp[,[exp][,[exp]]]) }
```

11. logical expression:

```
    [operator] operand [operator operand] [operator operand]
                                                        [...]
 where:
            ADD[C]
            AND
operator =  BOOLh        operand =  alpha-expression
            OR                      ALL function
            XOR
```

Alphanumeric Operations

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

- Alpha-expressions which are _not_ receivers are moved to new locations before being used or changed. Note that this includes alpha-receivers enclosed in _parentheses_.

- Except in the TRAN statement, described in Part II, alpha-receivers are _never_ moved; they are operated on in place. This can cause quite distinctive results, such as single-character propagation and boolean operations applied _stringwise_ to a receiver. The difference in results between alpha-expressions which are not receivers and those that are is especially noticeable in the following:

  ADD
  AND
  OR
  XOR
  BOOL
  COPY
  LET

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

2. TRAN _always_ moves the translation alpha to a separate translate table, inaccessible to the user. Thus, TRAN may _never_ translate its own table, accidentally or otherwise.

3. Statements which perform multiple assignments always assign values from left to right, except the LET statement itself, which operates from _right_ to _left_. This applies particularly to: 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.

Miscellaneous Terms

1. file-number: #int
   where 1 ≤ int ≤ 64

2. _file-expression_: #expression
   (or _file exp_) where 1 ≤ expression ≤ 64

3. _line-number_: digit[digit][digit][digit][digit][digit]

4. <u>receiver</u>: {alpha receiver }
{numeric variable}

5. <u>array-designator</u>: {alpha array-designator }
{numeric array-designator}

6. <u>d</u>: a digit (0,1,2,...8 or 9)

CHAPTER 4 CONTROL STATEMENTS

4.1 INTRODUCTION

In normal processing a VS BASIC program is 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. VS BASIC control statements are listed in Figure 4-1.

CALL	FOR...NEXT	INPUT
RETURN	IF...THEN...ELSE	ACCEPT
END	ON...GO TO	STOP
GOSUB	ON...GOSUB	Intrinsic Functions
GOSUB'	GOTO	Unusual Condition Exit Clauses
FN		

Figure 4-1. 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, or if encountered in a subroutine, returns program control to the calling program (see number 3). STOP temporarily halts execution until the program user presses the workstation ENTER key, or, under defined conditions, one of the program function keys.

*NOTE: A number of other BASIC statements also have error/data conversion exits.

INPUT and ACCEPT, like STOP, temporarily halt execution, in this case to enable the program user to supply the program with run-time data, or, once again under defined conditions, to press a program function key. These statements are discussed in Chapter 5 and under their separate entries in Part II.

2. Looping - A powerful feature of BASIC is its ability to execute repeatedly 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 II.

3. Program Branching - A number of statements direct program execution to a specified section of code. GOTO transfers control to the line number following the GOTO verb. GOSUB, GOSUB', and CALL transfer control to various kinds of subroutines, after the execution of which control can be returned to the main body of the program by RETURN or END. GOTO is discussed under its entry in Part II; subroutines are discussed in Section 4.2 and under the entries for the various statements in Part II.

   Related to subroutines are functions, sections of code which, when invoked, are given an argument and return a value. VS BASIC has both intrinsic and user-defined functions. Intrinsic functions are sections of code residing in the compiler. When an intrinsic function name is used in the source file, the code for that function is compiled into the object file and is invoked whenever the function name appears. The intrinsic functions include all the standard trigonometric functions (SIN, COS, TAN, etc.), a number of other numeric functions (for rounding, random number generation, etc.), a number of alphanumeric functions (for operation on substrings etc.), and four system functions for disk I/O. Intrinsic functions are discussed in Section 4.3 and under the entries for the various functions in Part II.

   User-defined functions are defined by a DEFFN statement and invoked by an FN function call. These are discussed under their entries in Part II.

4. 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 II.

5. 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 statements. These clauses in file I/O statements are discussed in Chapter 6, Section 6.7, and in the appropriate entries in Part II.

4.2 SUBROUTINES

A subroutine is a group of program lines which can be invoked from any point in a program any number of times 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. This feature gives subroutines a distinct value - 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 two main types of subroutines: internal and external. 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 depressing 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 DEFFN' statement. Subroutines invoked by GOSUB are not marked - GOSUB transfers control to a specific line number.

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 2200VS Programmer's Introduction, Chapter 4). The main program invokes external subroutines by means of the CALL statement. External subroutines have the advantage that they 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 changed, it has to be changed and recompiled only once, and the calling programs do not have to be modified.

4.3 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 II of this manual.

GOSUB

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

```
500 GOSUB 2000
```

When executed this statement transfers control to line 2000. The beginning of the subroutine is not specially marked. Any legal 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
```

The system 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 900, execution continues then at the statement following the GOSUB statement on line 900. 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.

GOSUB'

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

```
500 GOSUB'112
```

This statement will cause control to pass to the statement DEFFN'112. The range of allowable DEFFN' numbers is 0 to 255. (Note that, unlike with GOSUB, it is not necessary for the programmer to keep track of the line number when using GOSUB' subroutines.) 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. For example:

```
 500 GOSUB'112 (A$,B)
      .
      .
      .
2000 DEFFN'112 (D$,X)
```

This pair of statements does the following: implicit assignment statements are generated, assigning the current value of A$ to D$, and of B to X, thus passing these values to the subroutine. When the subroutine ends, however, the values of D$ and X are not passed back to A$ and B.

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 DEFFN' 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.

Program Function Keys

The VS workstation has, at the top of the keyboard, 16 Program Function (PF) Keys, each of which can be depressed in upper- or lowercase for a total of 32 Program Functions. BASIC can program any of the PF Keys to invoke marked subroutines.

Subroutines invoked from the keyboard, like those invoked with a GOSUB' statement, are marked by DEFFN' statements, with the restriction that the legal range of DEFFN' numbers for PF key accessible subroutines have numbers 1 to 32 (instead of 0 to 255). A DEFFN' 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 DEFFN' subroutine whose number corresponds to the number of that PF key. For example:

```
 500 STOP
      .
      .
      .
2000 DEFFN'1
```

Depressing PF 1 when execution is halted by the STOP at line 500 invokes the subroutine marked by DEFFN'1. Keying ENTER, however, causes the normal sequence of execution to continue with the statement following STOP.

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 statement following. Thus, DEFFN' subroutines can be invoked repeatedly from a STOP or INPUT statement.

4.4 EXTERNAL SUBROUTINES

Using the GOSUB, GOSUB', DEFFN', and RETURN statements discussed above, a program can transfer control to a portion of the program--called a subroutine--with the understanding that, when the subroutine has completed execution, control will return to the statement following the GOSUB or GOSUB'. A subroutine can be called from several points in the program, with control returning each time to the point from which it was called on that occasion. Subroutines defined in this manner are wholly contained within the source program file and are referred to as "internal subroutines."

There is a second class of subroutines which 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 a subprogram. Subprograms are distinguished by the fact that their first statement, other than REM, must be the SUB statement.

The reasons why external subroutines might be preferred over internal GOSUB or GOSUB' subroutines are as follows:

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. Thus, the SUB subroutine is BASIC's primary interface to other languages such as COBOL and Assembler.

External subroutines are the reason for the linking stage of program development. If a program is contained in several files (a main program and several subprograms), the linker must be used to merge the respective object files produced by each compilation into a single object file which is executable by the VS.

Operation of External Subroutines

The CALL statement transfers control from one program (called the calling program) to the beginning of another program (the external subroutine). The point at which the CALL statement occurs in the main program is saved, so that control may later return to that point. The SUB statement declares a program to be a subroutine, allowing it to be named in CALL statements. A subroutine may, itself, contain one or more CALL statements, by which it calls other subroutines. A subroutine cannot, however, 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. Then control returns to the point in the calling program from which the subroutine was invoked on that occasion--i.e., control returns to the statement following the last CALL statement executed.

The apparent effect of a CALL statement in a calling program is to invoke 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.

Passing Values to Subroutines

A trivial example of a subroutine is a process which adds two to a number, and returns the result. It is clear that a method must be used to pass from the calling program to the subroutine the number which is to have two added to it, and to allow the subroutine to pass the result back to the calling program. In BASIC, these numbers are passed in one of two ways:

1. The numbers may be made arguments or operands of the subroutine. An operand of a subroutine means simply a number which may be operated on by action of the subroutine. These numbers are enclosed in parentheses following both the CALL and the SUB statements. The arguments which follow the CALL statement define the items which the subroutine will operate on. The arguments of the SUB statement indicate the names by which each respective item will be referred to in the subroutine. There is a firm, one-to-one correspondence between the CALL arguments and the SUB arguments, based on their position in the argument list.

   To overcome the problem which can be created when BASIC modules call or are called from non-BASIC modules, two forms of SUB and CALL are provided.

A. <u>non-ADDR form</u>

This is the standard BASIC argument-passing scheme, which passes/accepts the "dope vectors" constructed for arrays and alpha-expressions/receivers. With this form, any dimensions/lengths specific within the SUB program are ignored, since they are specified by the dope vectors. Only the vector/matrix/scalar distinction is significant.

B. <u>ADDR form</u>

This form is 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: ADDR form for CALL causes all argument-passing to be done via pointers to the actual values; dope vectors are <u>not</u> 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.

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

2. The items may be made <u>common</u> (or "placed in common"). The programmer places several variables, which are usable as any other variables are, in a COM statement. This statement essentially declares that the compiler is to place these variables in memory in an area which is accessible to all subroutines. A COM statement may appear in a subroutine to indicate that named variables reside in the common area. Thus, if a subroutine changes the value of any variable in the common area, any other subroutine or calling program may detect the change by examining the common area.

There is a correspondence similar to the correspondence in CALL/SUB arguments, since there is actually only one common area. Although calling programs and subroutines may use different variable names to refer to variables in the common area (as specified in each COM statement), they will be accessing the same variables.

Here is an example, assuming the first method (passing arguments) is to be used. The calling program calls the subroutine which will add two to a number. The number to be incremented--in this example, X--is listed as the argument of the call:

```
14400 CALL "ADDTWO"(X)
```

Control is transferred to the subroutine called ADDTWO. The SUB statement in that file declares the name of the subroutine, and indicates that it will call the first argument "I".

```
100 SUB "ADDTWO"(I)
```

Thus, on this call, the variable I refers to what the calling program calls X. The statement:

```
200 LET I=I+2
```

adds two to the variable called X in the calling program. When the END statement in the subroutine is reached,

```
300 END
```

control returns to the calling program (specifically, to the statement following line 14400).

On a subsequent call, the calling program might give the subroutine the same or a different variable as the argument.

```
20600 CALL "ADDTWO"(E)
```

The subroutine would use the variable name I this time to refer to the calling program's E, and would add two to it.

The preceding example could have used common instead. The calling program declares that it will use the first location in the common area, and will access it as the variable X.

```
100 COM X
```

Later in the calling program, the call to ADDTWO occurs. In this situation, no arguments are required.

```
1440 CALL "ADDTWO"
```

The subroutine must declare that it too will use the first location in the common area, and must declare the name it will use to access that location.

```
100 SUB "ADDTWO"
200 COM I
300 I=I+2
400 END
```

Thus, by adding 2 to I, the subroutine adds 2 to the same variable which the calling program has called X.

Unless linkage between variables in the calling program and variables in the subroutine is made in one of these two ways, any variable name may be used in a subroutine, independently of any other usage of it in the calling program or another subroutine. Likewise, line numbers in a subroutine do not correspond with or interfere with line numbers in another file. The GOTO statement, for instance, cannot be used to transfer control from file to file. Interaction between programs and external subroutines occurs only in these two ways.

Argument Types

As the example showed, 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. Thus, the variable name listed first in the parentheses in the SUB statement will be the name used by the subroutine to refer to the first argument passed by the CALL statement. The second variable name will be linked to the second argument in the CALL statement, and so on.

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

100 SUB "SUBROU" (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.

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, contrary to normal procedures, 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.

Notice that an array string 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.

NOTE:

> Array strings longer than 256 bytes will be truncated.

The number of subscripts associated with a variable must be consistent from the calling program to 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 as either a vector or a matrix. (If an array whose designator appears in the SUB statement does not appear in a DIM statement, it is assumed to be a matrix.) In the DIM statement, the supplied dimensions are irrelevant; the actual upper limits are taken from the array as dimensioned in the calling program. In fact, a MAT REDIM statement may occur in a subroutine, and the redimensioning of the matrix will remain in effect when control returns to the calling program.

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. However, unless linkage is made in this manner, any files 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.

Programming Considerations

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

No variable name occurring 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.

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 " ", integer variables are initialized to 0%, and floating-point variables are initialized to 0. 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 "ABCDEF"(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
```

CHAPTER 5 WORKSTATION AND PRINTER INPUT/OUTPUT STATEMENTS*

5.1 INTRODUCTION

VS BASIC contains a group of statements designed 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.*

The statements intended purely for data output are:**

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.

PRINT — used to print data on the printer or display data at the workstation, one line at a time. The output mode is determined by a SELECT statement (see Part II). The data can be directed to specific positions on the workstation screen, but it cannot be formatted. The screen is not cleared before the data is displayed.

* VS BASIC also supports the creation of printer files using SELECT, OPEN, WRITE, and CLOSE. See Chapter 6, and the descriptions of the individual statements in Part II.

** VS BASIC also includes a specialized statement for dealing with arrays, MATPRINT, which is used to print or display the contents of arrays on the printer or at the workstation without specifically naming each element.

PRINTUSING — used to print or display a formatted output line on the printer or at the workstation. The format to be used is supplied by a FMT or % (Image) statement. The data cannot be directed to specific positions on the workstation screen, and the screen is not cleared before the data is displayed. The output mode is determined by a SELECT statement (see Part II.)

The FMT and % (Image) statements are nonexecutable statements which contain formatting information for a PRINTUSING statement. (FMT and % (Image) statements may also be used to specify a format for input/output with disk files. See REWRITE, WRITE, READ, GET, and PUT in Part II.)

All the VS BASIC input statements can also be used to some extent to output data or messages. None of this output, however, can be directed to the printer. Two statements are capable of outputting a one-line message, as well as receiving data. The message and data cannot be formatted using these statements, and the message and data cannot be directed to specific positions on the workstation screen. Both of these statements halt program execution to permit a response by the user. These statements are:*

INPUT — used to receive data entered from the keyboard on a line-by-line basis. A message can be inserted before the question mark which INPUT automatically displays.

STOP — used to stop execution of the program until some user action is taken. A message can be output, and data can be entered and passed to a DEFFN' subroutine.

* VS BASIC also includes a specialized statement for dealing with arrays, MAT INPUT, which is used to sequentially input the elements of an array without naming each element separately. See Part II for more information.

One additional input statement is capable of generating formatted displays, of using FAC's (see Section 5.5) to affect the appearance of the displayed data, as well as validating the input data, and controlling program execution. This statement is:

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. ACCEPT can validate the data according to a given range and, through the use of FAC's, can validate according to the data type. In addition, ACCEPT can:

1. disable and enable PF Keys and the ENTER key and use them for limited data entry, or for controlling program execution,

2. provide an exit if no new data is entered,

3. display the current values of the data which are to be altered, and pseudoblanks if there are any blanks or spaces combined in the data, and

4. control the appearance of the data through the use of FAC's.

ACCEPT cannot explicitly access DEFFN' subroutines or character strings, or direct output to the printer.

The workstation and printer I/O statements are summarized in Table 5-1.

This chapter will discuss the concepts of the VS workstation screen and printer before going on to the individual statements.

Table 5-1.

Summary of Workstation/Printer I/O Statements

	accepts data input			output only	
	INPUT	ACCEPT	DISPLAY	PRINT	PRINTUSING
Data verification		0			
Displays pseudoblanks		0			
Clears screen before execution		0	0		
PF key data entry and execution control	0	0			
Can format data		0	0		0
Displays data at a given position		0	0	0	
Displays current values		0	0	0	0
Access to printer				0	0

5.2 PRINTER OUTPUT

The PRINT and PRINTUSING statements can output to a printer as well as to the workstation screen. Either literals or the current value of any variable or expression may be output, using a wide variety of formats. 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.

The PRINT and PRINTUSING statements actually move data to a line buffer for the printer. The contents of the line buffer are printed only when an implied or explicit move to the next line occurs, e.g., via SKIP or via a PRINT statement with no trailing semicolon, or when data overflows the capacity of the line buffer. When the buffer has been printed, it is cleared and enabled for re-loading beginning 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.) The program may also cause an arbitrary number of blank lines to be fed from the printer. However, data may not be erased from the printer once it is printed, nor may a line which has been sent to the printer be recalled for modification.

The "wraparound" concept (see Section 5.4) is valid for the printer as well as the screen. 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.

5.3 PRINTUSING, FMT, AND IMAGE STATEMENTS

The PRINTUSING statement (and a number of disk I/O statements) uses an auxiliary statement to define the format of the data to be input or output. The line number of the auxiliary statement is written after the USING clause. The statement at this line number must be either a FMT statement or a % (Image) statement. For instance:

```
15600 PRINTUSING 15700, list of expressions
15700 FMT list of specifications
```

The position of the FMT statement with respect to the statement with the USING clause is irrelevant. PRINTUSING statements may contain a list of expressions which are to be output. FMT and Image statements contain (among other things) "data specifications" which specify the format for outputting a single expression. Starting at the beginning of both lists, items from the list of the PRINTUSING statement must correspond with the data specifications in the FMT or Image statement: If the FMT or Image statement contains a string specification, the corresponding item in the PRINTUSING statement must be of string type. For example:

```
1400 PRINTUSING 1500,M,N
1500 FMT PIC ($#####),PIC(####)
```

prints the current value of M using the specification PIC($#####), and then prints the current value of N using the specification PIC(####).

If there are more items in the PRINTUSING 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 PRINTUSING statement. Note that an error message will be produced if a PRINTUSING statement with a non-null argument list is used in conjunction with a FMT or Image statement which contains no data specifications. In PRINTUSING, subsequent output will occur on the next line down unless the item in the PRINTUSING 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 PRINTUSING statement, then 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 PRINTUSING. This situation can also occur when a FMT or Image statement is reused, but contains more data specifications than there are items remaining in the PRINTUSING statement.

```
1100 %-### XYZ -###.##
1400 PRINTUSING 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 and remains in the PRINTUSING list, but the Image statement is exhausted. Therefore, it 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.

5.4 THE WORKSTATION SCREEN

The workstation display contains 24 rows of 80 characters each, for a total of 1920 positions capable of holding characters. Each character position in the display can be referred to by its row and column number. Thus, 1,1 is the first position on the top row; 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.

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. A "wraparound" concept exists: 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 remainder of the current line, and the rest will be displayed on the next line down. Note, however, that column 80 of line 24 (the "end of the screen") does not wrap around to column 1 of line 1.

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 the cursor is 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, 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.

5.5 FIELD ATTRIBUTE CHARACTERS

Any position on the screen may contain any 8-bit binary code. The codes from HEX(00) to HEX(7F) represent characters which can be displayed on the workstation screen. HEX(20) is the character "space" or "blank"; it occupies a character position, but causes nothing to be displayed there. HEX(00) also displays as a blank.

The codes from HEX(80) to HEX(FF) are Field Attribute Characters (FAC's). FAC's also occupy a character position, but do not display a graphic character. FAC's 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 bright, dim, blinking, or nondisplay (i.e., displayable characters of the field will be suppressed). These four options are mutually exclusive.

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

3. Whether the field is modifiable by operator input, or not modifiable (protected).

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 input; any characters can be output in any field type. This information is irrelevant if the field has already been declared "protected" by option 3.

Appendix C contains a listing 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 with those characteristics to the immediate left of column 1 of each line.

The programmer may output FAC's 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, protected, with no underlining, and all characters displayed.

The INPUT statement places a FAC (of HEX(81)) in the screen buffer to the left of the field where input is designated 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)). Thus, a % sign may not be input to indicate an integer. The programmer may override these FAC values by making an explicit specification in the ACCEPT statement. (This is discussed in Section 5.7.)

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

5.6 DISPLAY

DISPLAY formats the entire workstation screen for data display. The programmer using DISPLAY should bear in mind that 1) the entire screen is cleared prior to the display of data, and 2) the entire screen is therefore available to be written upon.

Two points arise from these considerations. First, the programmer has much more reason to control the line and the column at which individual data items are displayed. As a result, the AT clause (which is also available for PRINT) is a particularly helpful tool for DISPLAY. For example, the following statement:

```
DISPLAY     AT (5,10), "RESULTS=",A,                    !
            AT (7,10), "EXPECTED RESULTS=",B
```

results in a screen with nothing but the data indicated above displayed in the stipulated positions. The user should note the use of the continuation character ("!"). Because DISPLAY formats an entire screen, it is often impossible to get all the operands onto one program line. In fact, it is a good practice to use one program line of a DISPLAY statement for each screen line.

Second, since DISPLAY works with the entire screen, line-oriented format statements used with PRINTUSING -- % and FMT -- cannot be used with DISPLAY. Consequently, format information is specified within the DISPLAY statement itself through PIC and CH clauses, which work like those in FMT.

Note that with the exceptions of the BELL and COL keywords, the syntax and functions of DISPLAY are strictly included in the syntax and functions of ACCEPT.

5.7 ACCEPT

The ACCEPT statement formats the entire screen for data input, first erasing everything on the screen. The significant differences between ACCEPT and INPUT are listed in the table below:

Table 5-2.
Differences Between ACCEPT and INPUT

ACCEPT	INPUT
Entire screen cleared	Screen not cleared
Entire screen utilized	Only uses as many lines as necessary
Current data values displayed	No data values displayed
Generates pseudo-blanks	No pseudoblanks
FAC usage permitted	No FAC usage
CH and PIC clauses	No CH and PIC clauses
Range verification	No range verification
DEFFN' text strings cannot be explicitly called through PF key usage	DEFFN' text strings can be directly called by striking PF keys
Data entry key can be specified	Data only entered through ENTER key
Can alter execution if data is not altered.	Cannot determine if data has been altered.

ACCEPT is the most complex of the screen I/O statements available in VS BASIC. Its various options, however, need not all be used in any given statement. The following discussion of ACCEPT breaks the statement down into its three main features -- screen handling, data validation, and PF key usage -- and treats the clauses pertaining to each of them one at a time.

Screen Handling

AT

As mentioned before, ACCEPT works with the entire CRT screen. The programmer, consequently, can position data items where desired, or use default values provided by ACCEPT. Data items can be explicitly located through AT, in the same manner as PRINT or DISPLAY. Literals can be displayed by ACCEPT as well as variables; one significant difference between INPUT and ACCEPT is that ACCEPT can position any number of literals anywhere on the screen through the use of AT clauses.

> **NOTE:**
>
> If a field is placed so that it would overflow the end of a line, it will be displayed entirely beginning at the second byte of the next line. This is true for ACCEPT and DISPLAY. PRINT, however, will print beginning at the assigned position, and only the overflow will appear on the following lines.

FAC

ACCEPT allows the use of Field Attribute Characters (FACs) to modify the way in which data can be displayed. The FAC clause lets the programmer control the display mode of variables. The following statement:

```
100 ACCEPT AT (5,1), FAC(HEX(8C)),A$                    !
           AT (6,1), FAC(HEX(90)),B$
```

causes A$ to be displayed dim and protected, while B$ is displayed blinking and modifiable. This gives the programmer control over what is to be modified.

Further control over Field Attribute Characters comes from the ability of FAC to take alpha-expressions as operands. Thus, the following legal statement:

```
100 ACCEPT AT (5,1), FAC(Z$),A$
```

means that the mode in which A$ is displayed depends upon the value of Z$ at the time of execution.

The default FAC for a receiver field is bright and modifiable; for any other field it is dim and protected.

In addition, the default FAC for a receiver field is tabable (i.e., the TAB Keys can be used to arrive at the beginning of the field), and non-tabable for any other field. The protected numeric FACs (see Appendix C) are always tabable and can be used to create a tabable, non-modifiable field.

CH, PIC

Like DISPLAY, ACCEPT allows variables to be formatted within the statement, using CH and PIC clauses in the manner of FMT.

Data Validation

Like INPUT, ACCEPT performs an implicit validation on entered data based upon the FAC preceding the field which receives the data. A numeric field (e.g. a field whose corresponding receiver is a numeric variable) will not accept alphabetic data, and so on. An attempt to enter illegal characters causes the ACCEPT screen to be redisplayed, with the cursor at the beginning of the first illegal field.

RANGE

In addition to its implicit verification, ACCEPT allows the user to set a range of acceptable input, against which the program validates data before accepting it. The format is as follows:

```
ACCEPT A, RANGE (0,100)
```

The operands of RANGE set the limits of acceptable values of data. If the user attempts to supply data outside that range, the data will not be accepted by the program, and the field will be set to blinking.

RANGE can be set for alpha variables as well as for numeric. In the case of alpha variables, the program uses the ASCII collating sequence.

ALT, NOALT

The program can determine if the data has been modified. Any keyboard action that is performed on the field, even retyping the old data or erasing pseudoblanks, will indicate to the program that the data has been altered. This feature is used to perform two different functions.

ALT

Ordinarily all modifiable fields are read, validated, and transferred to their receivers, whether or not the fields were actually changed by the user. Inclusion of the ALT clause causes the program to read, validate, and transfer only the fields which actually are modified.

NOALT

The NOALT clause (NOALT GOTO or GOSUB and a line number) specifies that program control be transferred to the indicated line number if none of the values in the accept statement are altered. The functions of the ALT clause are also included in NOALT.

> **NOTE:**
>
> A single ACCEPT statement _cannot_ include both ALT and NOALT.

PF Key Control

ACCEPT has three clauses that allow PF key control: KEY, KEYS, and ON key GOTO. Note that PF keys do not call DEFFN' subroutines or strings.

KEYS

This clause specifies the keys which are valid for a given ACCEPT; any others will cause a beeping sound if pressed. KEYS must be followed by an alpha-expression (normally in the form of HEX(xxxxxx..)), which is used as a list of one-byte HEX values corresponding to the allowed PF key. The ENTER key is identified by HEX(00), PF 1 by HEX(01), and so on to PF 32, which is HEX(20).

Thus, to allow PF 1 and PF 2 as the only acceptable keys, the KEYS clause would be as follows:

```
KEYS(HEX(0102))
```

KEY

The KEY clause assigns the number of the key depressed to a numeric receiver. For example,

```
KEY(N)
```

means that the value of the key depressed will be assigned to N. If the ENTER is struck, N equals 0; if PF 1, N equals 1; etc. Note that only keys made legal by the KEYS clause can be used for KEY.

ON Key Value

This clause allows the user to exit without changing any data values if certain PF keys are pressed.

CHAPTER 6 DISK, TAPE, AND PRINTER FILE I/O

6.1 INTRODUCTION

A VS BASIC program is capable of reading and writing both consecutive and indexed (including alternate indexed) files. The statements that support this capability are shown in Figure 6-1.

SELECT	DELETE
OPEN	SKIP
READ	PUT
WRITE	GET
REWRITE	CLOSE

Figure 6-1. Disk I/O Statements

This chapter contains a general discussion of the concepts of file I/O in VS BASIC. A detailed discussion of each of the statements can be found in Part II.

6.2 FILE HIERARCHY

The creation and maintenance of files is controlled by the VS Data Management Subsystem. A "file" is a logical unit consisting of one or more records. A file may contain source program text (a "source file") or object program code (a "program file"), or it may contain data records. Files can be opened and named by the user; Data Management automatically handles the complex "housekeeping" chores associated with creating and maintaining an external file. Each disk file is located within a hierarchical structure consisting of two higher levels: libraries and volumes; each tape file is located within a volume only (tape volumes do not have 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 source, program, and data files, or special libraries may be designed 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. Finally, duplicate volume names are allowed but not recommended.

To avoid possible ambiguity, each file name must be qualified with the names of its associated library and volume when the file is opened. Such qualification is not required, however, when running programs from the system program library, because the system library and volume are used automatically whenever the named program is not located in the user program library, or if no user library is supplied. Because all system utilities are stored in the system program library, it is never necessary to specify a library or volume name when invoking a system utility program.

Libraries and files have names which can contain up to eight characters. Volume names contain up to six characters. Each of these names must begin with an uppercase letter or one of the special characters $, #, or @; characters after the first may be any alphameric character including the special characters. (Embedded spaces are not allowed.) In order to open a new file, the system requires the name of the file and library, or file sequence number for tape files, which will be used in the future to refer to the new file, and the name of a volume which is, or can be made, accessible to the VS. The combination of the file and library names, or the file sequence number, must be unique on that volume. If an old (already-existing) file is to be opened, the names of the file, library, and volume must correctly identify the old file; that is, they must match the names used to create that file. Similarly, an old tape file must be identified by the file sequence number used when it was created. The volume name is unimportant for tape.

In a BASIC program, the names of the file, library, and volume are used only once, in the OPEN statement, which enables input and/or output with that file. (BASIC automatically assigns a file sequence number of 1, but this can be changed at run time.) The OPEN statement associates the named file with a file number (which has nothing at all to do with a file _sequence_ number). File numbers are preceded by a pound sign (#), and can range from 1 to 64. After a file is OPENed, the file number is used instead of the file's name to refer to the file. Thus, a program can be rewritten to access a different file by simply changing the OPEN statement to associate a different file name with the same file number. An expression can be used as a file number (e.g., #N). In this case, the file numbers can be changed dynamically during execution, thus enabling a single statement to reference several files in the course of program execution. For example, #(X+2) refers to file number #3 if X=1, but refers to file number #1 if X=-1, and so on.

The final level in the disk hierarchy is the record. A record is analogous to a line on a piece of paper. Information is read from or written to a file one record at a time.

Tape files, however, are made up of blocks which in turn are divided into records.

6.3 SELECTING FILE NUMBERS (SELECT)

At the time a program is compiled, the compiler must know how many file numbers will be used in the course of the program, so that it can include as part of the compiled program a User File Block (UFB) for each file number used. For instance, in a program where #1, #2, and #3 were going to be used, the compiler would have to produce an object file containing not only the program, but also three UFB's. To indicate this to the compiler, the BASIC programmer must include exactly one SELECT statement in his program for each file number he intends to use. The SELECT statement directs the compiler to allocate a UFB.

Certain characteristics of a file must be known in order to set up an appropriate UFB. These characteristics are specified in the SELECT statement; if a file number will be used for more than one file during the course of the BASIC program, all files must be identical with respect to those characteristics.

A disk file may be constructed so that the records are consecutive (sequential), that is, they appear in the order in which they were written, in which case the SELECT statement contains the word CONSEC. Alternately, the file may be indexed so that the records are ordered according to the contents of some part of the records, called a "Key". In this case, the SELECT statement contains the word INDEXED and also specifies the location and length of the Key. Indexed files may also have alternate indices, that is, additional keys which can be used to access the records in the file.

A tape file can only be written consecutively. The SELECT statement contains the word TAPE, and the size of the BLOCKS into which the tape file is divided.

The records in a disk or tape file may be of a fixed length (in which case the length must be specified) or variable length (in which case the programmer writes VAR, and must specify the maximum legal length). If the records are variable length, they may be "compressed" (in which case the programmer writes VARC). Compressed records are expanded before being made available to the BASIC program, and recompressed before being rewritten to disk. Using compressed records conserves disk space, and is transparent to the user.

Additionally, a PRINTER file may be created. This special type of file can only be written to (i.e., records <u>cannot</u> be read), and is identified by the word PRINTER.

A file number may be used to refer to a single file, or to several files whose characteristics are identical over the course of the program. A file number may be used to refer to only one file at any given time. A file which is "open" on a file number (by virtue of having been referenced in an OPEN statement) must be CLOSEd before another file can be OPENed using the same file number.

When the SELECT statement is written, a parameter reference name ("prname") must be assigned to the file number. If any OPEN statement for that file number fails to provide a valid file, library, or volume name, or opens a file whose characteristics conflict with those specified in the SELECT statement, the VS Data Management System will, at the time that OPEN statement is executed, ask the user to specify the information which was not supplied by the program (the user will always be asked for the appropriate information at run-time in the case of tapes). Data Management will use the prname, not the

file number, to indicate to the user which file number needs additional specification. Note that this will also occur with valid parameters if neither NODISPLAY nor NOGETPARM is included in the OPEN statement.

This means that a program can be written to operate on the data in a file, but that the name of the file need be known only at the time the program is executed. On different executions, the operator may supply a different file name, allowing the program to operate on different files with no reprogramming at all.

6.4 OPENING A FILE (OPEN)

The OPEN statement enables input or output between a BASIC program and a file. The CLOSE statement severs this connection. Thus, input and output, using the READ and WRITE statements, for instance, may occur only while the file is "open." The OPEN statement, must specify the mode in which the program will use the file:

1. It can open for INPUT a file which already exists. The program will then be able to read from the file, but not modify it. (Does not apply to printer files.)

2. It can open for I/O (Input and Output) a file which already exists. 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 may do REWRITES in I/O mode, but may not create new records. (Does not apply to printer or tape files.)

3. It can open a new file 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.

4. It can open a preexisting file (consecutive only) for EXTEND. The program will then be able to write to the file, but not read from it. The first record written will take its place directly following the last record which was already in the file. (Does not apply to printer files.)

5. It can open a preexisting file SHARED. This mode is similar to I/O mode, but allows simultaneous access to the file by other VS users. (SHARED mode is supported only for INDEXED files and for a special type of consecutive file called a "log file".)

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 for 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.

6.5 SUMMARY OF I/O STATEMENTS

Associated with each file number is a data buffer, which serves as an intermediate storage location for data transferred between BASIC variables and the disk or tape. A READ statement reads one record of the file into the buffer, and may then transfer information from there into one or more variables in the BASIC program. A WRITE (or REWRITE) statement transfers information from the buffer out to a file, typically after loading the buffer from one or more variables.

Data may be written to disk or tape either in BASIC's internal format, or using explicit formatting. Likewise, data appearing on disk in either BASIC's internal format, or another format such as decimal ASCII or packed decimal, can be read into receivers in the BASIC program. If explicit formatting is to be used rather than BASIC's internal format, the keyword USING is specified in the file I/O statement, followed by the line number of a FMT or % (Image) statement which defines the formatting to be used. (The FMT and % (Image) statements are described in Chapter 5, Section 5.3.)

The PUT statement transfers data from BASIC variables into the buffer (or any alpha-receiver). The GET statement transfers data from the buffer (or alpha-expression) into BASIC variables. In both statements, a conversion between the internal BASIC format and another format may occur. PUT is like WRITE, and GET is like READ, except that neither PUT nor GET actually does I/O to the disk or tape, but only between the program and the buffer.

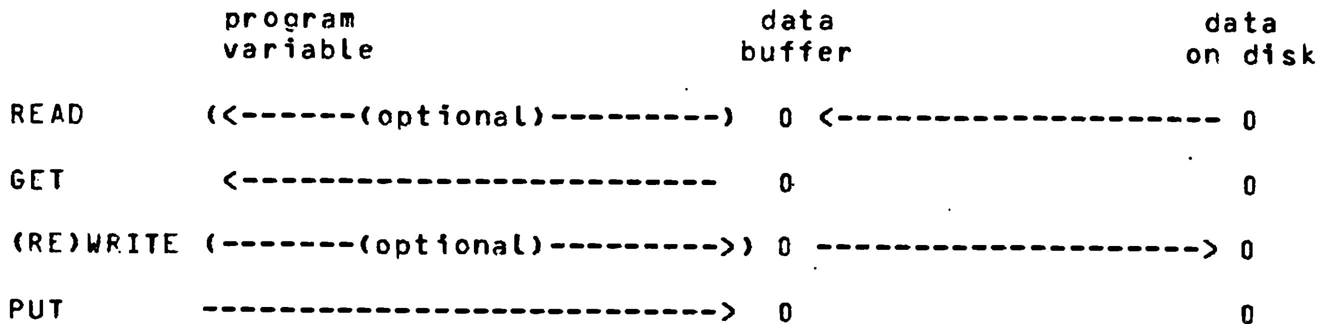


Figure 6-2. Information Flow in the File I/O Statements

By omitting a list of receivers, the programmer may specify that a READ statement simply brings a record into the data buffer. The programmer must then use a GET statement to extract the data from the buffer. The GET statement allows the programmer to extract the contents of the same file record more than once. This might typically be done reading the values in using different Images (formats), or reading the values into different variables the second time.

Similarly, by omitting the arguments of the WRITE statement, the programmer causes the data buffer to be written into the file. The assumption is that previous PUT or WRITE statements were used to load meaningful information into the data buffer. Multiple WRITE statements using the same buffer can be used to write multiple identical records out to the file without having to undergo a redundant format conversion each time.

The remaining file I/O statements are more specific in their function. In CONSEC files, where records are organized in sequential order, the SKIP statement allows the program to move forward or backward a specified number of records, or to return to the beginning of the CONSEC file.

When a record from a SHARED or I/O file is read and the HOLD option is specified, subsequent use of the REWRITE or DELETE statements allows the programmer to rewrite or delete records.

6.6 FILE I/O SYSTEM FUNCTIONS

Four functions may be used in expressions to retrieve information concerning file I/O operations: FS, KEY, MASK, SIZE.

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) No more room in the file (OUTPUT or EXTEND mode)
"30"	Hardware error
"95"	Invalid function or function sequence
"97"	Invalid record length

KEY(file-expression)

The KEY function returns the "key" field from the specified file's data buffer. The file-expression given as the argument of the KEY function must refer to an INDEXED file. 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 exactly 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"
```

This use of KEY is typically performed immediately preceding a WRITE or REWRITE statement. Note that 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=" construction.

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 the HEXOF, the result of the MASK function.

Example: Having read a record with seven alternate keys,

```
400 DIMA$2
500 A$ = MASK(#1)
600 PRINT HEXOF (A$)
```

Result: FE00

which represents the binary string 1111111000000000, which indicates that the first seven alternate keys are used in this record.

SIZE(file-expression)

The SIZE function returns as an integer the size in characters of the record most recently read from the specified file.

6.7 ERROR RECOVERY

The situations under which an Input/Output instruction cannot be successfully completed fall into four categories:

1. Errors handled by the 2200VS 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. Type 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 6.6).

3. Type 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 ###.

4. Type 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 by means of a dialogue with the workstation operator at the time of the error. BASIC allows the user to specify the program line numbers to go to 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 servicing, the programmer specifies three things: (1) the type of error situation to be covered; (2) the type of transfer of control to be performed--i.e., returning or nonreturning; and (3) the BASIC line number to which control is to be passed. For instance, to force a returning branch to line number 003300 if a data conversion error (whose symbol is DATA) occurs, the programmer writes:

```
DATA GOSUB 3300
```

in the selected READ or WRITE statement.

Service routines 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. Service routines 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. Service routines 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 service routine for DATA type errors. The SKIP statement can specify an error service routine 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 service routine, 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 named file number, to determine the exact cause of the error.

CHAPTER 7 SPECIAL STATEMENTS: MATRIX AND DATA CONVERSION STATEMENTS

7.1 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 II.

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 BOOL (discussed in Chapter 3), the alphanumeric functions BIN and VAL (discussed in Chapter 4), and the binary arithmetic operation ADD (discussed in Chapter 3).

7.2 MATRIX STATEMENTS

> **NOTE:**
>
> The VS allows implicit redimensioning of alpha arrays in MAT =, TRN, and Sort Statements; the 2200T and the 2200VP do not. However, it is recommended that the matrices be explicitly (re)dimensioned to the same dimensions <u>before</u> using any of these statements, since future extensions may eliminate or change the implicit redimensioning as it currently exists.

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 II.

<u>I/O</u>

† 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.

<u>Assignment</u>

† 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.

† Statements marked with a dagger allow explicit redimensioning of arrays.

† MAT ZER	sets every element of an array to zero.

<u>Arithmetic</u>

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.

<u>Other</u>

MAT A or D SORT	sorts one alphanumeric or numeric array in ascending or descending order into a second array.
† MAT REDIM	redimensions an array.

† Statements marked with a dagger allow explicit redimensioning of arrays.

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.

<u>Array Dimensioning</u>

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 10 by 10, with a default alphanumeric length of 16. Each dimension may range from 1 to 32,767 with an alpha length 1 to 256.

The dimensions of an array may be changed explicitly during the execution of MAT statements 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 REDIM
MAT ZER
```

Arrays may also be redimensioned implicitly, as shown in the following example.

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.

Example:

```
MAT REDIM A$(2,3)10
```

This statement 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 number (or fewer) elements. For alphanumeric arrays, there must be the same number (or fewer) total characters.

Matrix Statement Rules

Certain rules must be followed in using 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 illegal. The same result can be achieved by using two MAT statements: MAT A = B+C, 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 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 legal MAT statements;
   MAT C=C*B and MAT B=TRN(B) are not.

PART II

BASIC KEYWORD FORMATS

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,

```
                  {literal       }
    operand =     {alpha variable}
                  {expression    }
```

   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 as needed can be added to the INPUT statement.

6. The order of parameters shown in the general form must be followed.

ACCEPT

```
General Form:

ACCEPT   list        [,list]...

       [,KEYS(alpha-arg1)] [,KEY(numeric variable)]

       [,ON alpha-arg2 { GOTO  } linenumber[,linenumber]...]
                       { GOSUB }

        [{ALT                          }]
        [                               ]
        [{NOALT {GOTO } line number}]
        [{      {GOSUB}                }]

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 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, all characters, tabbable (HEX(80))
Floating-point	- bright modifiable, uppercase, tabbable (HEX(81))
Integer	- bright modifiable, numeric only, tabbable (HEX(82))

2. The first character of the alpha-expression is used as the FAC character.

Field Placement Order

1. For single receivers, the fields are put out one at a time in order of appearance in the statement.

2. For arrays, the fields are put out element-by-element, in the usual row-by-row order (like MATPRINT).

Field Positioning

A field can either be explicitly placed at a specified row and column on the screen, using the AT clause of the ACCEPT statement, or if no AT clause is given it 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 which contains the preceding field will be placed beginning at 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 which is 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 beginning at the second position on the following line. If it is longer than 79 bytes, it is placed immediately following the preceding field, and it 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 tabable.

   These rules are summed up in Table II-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. If any two fields overlap.

2. If any modifiable field is longer than 79 bytes (too long to fit on a single line.)

3. If any explicitly positioned modifiable field extends beyond the end of the line on which it is placed.

4. If any field is explicitly placed so that it starts beyond the boundaries of the screen.

5. If any field extends beyond the end of the last line on the screen.

For arrays, an automatic "new line" (to column 2) is generated after each row.

TABLE II-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

Both numeric and alphanumeric fields may be validated by the BASIC program 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 only
   NEG = negative values only
   exp4, exp5 = lower and upper limits, respectively, for the input value(s) (inclusive).

2. Alphanumeric

   RANGE: alpha exp1, alpha exp2 = lower and upper limits. The ASCII collating sequence is used.

A second type of validation (specified by the FAC preceding a receiving field) is distinguished from RANGE validation by not setting the field to "blinking."

PF Key Control

(Note: PF keys in ACCEPT statements do not call DEFFN' subroutines or strings.)

The ENTER and PF keys can be controlled by any combination of three Key control clauses.

If all three clauses are omitted, only the ENTER key can be used to respond to the ACCEPT. If any one of the clauses is present, ENTER and all PF Keys are allowed by default, subject only to the restrictions of the KEYS clause if present.

1. KEYS

   This clause specifies the keys which are valid for this ACCEPT; any others will beep if depressed. 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. (Note that PF32 = HEX(20) may be considered to be a trailing blank if the user is not careful.) The key order is irrelevant.

2. KEY

   This causes the number of the Key (ENTER = 0) depressed by the user to be placed in the numeric variable. This is done prior to any field validation or exit branching. Note the KEYS clause takes precedence over the KEY clause.

3. 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 to which the program branches if that PF key is depressed.

   The ON clause is roughly equivalent to the statement

```
ON POS(alpha-exp=PF Key number){GOSUB}{GO TO}line number ...
```

   except that the actual length of the alpha-expression is used. Note that the KEYS clause takes precedence over the ON clause; thus, ON will never process invalid key values.

   Note also that the last line number should not be followed by a comma, nor should "omitted" line numbers be specified.

Use of Modified-Data-Tag

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. If either ALT or the NOALT exit is present in the ACCEPT statement, this 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.

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 depressed, the key is first checked for validity. If invalid, the workstation emits a beeping sound, and the user may continue modifying or he may depress 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. Note that 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 his mistake (and can further change other fields).

Examples:

```
100 ACCEPT "YOUR NAME",A$,B$
200 ACCEPT AT(10,10),"WHY DO YOU NEVER SPEAK?",FAC(HEX(90),!
300 A$,FAC(HEX(90),B$
400 ACCEPT A,B,C, KEYS(HEX(00010203040510)), KEY(A),ON    !
500 HEX(10) GOTO 1000, NOALT GOTO 2000
```

ADD [C] Operator

```
General Form:

 [LET] Alpha-Receiver = [logical exp] ADD[C] logical exp


   logical exp - see Section 3.8
```

Purpose:

The ADD operator is used to add a binary value to the binary value of an alpha variable.

```
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$; then, the value of B$ is 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 last (rightmost) byte of the value of the operand is added to the last (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 = 0122
```

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.

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
```

Examples of Valid Syntax:

```
100 A$=ADD HEX(FF)
200 A$=ADDC ALL(FF)
300 STR(A$,1,2)=B$ ADDC C$
```

See Chapter 3, Section 3.8 for more information on logical expressions.

ALL Function

```
General form:

    ALL (alpha-expression)
```

The ALL function has defined length equal to that of the function's receiver, and consists entirely of characters equal to the first character of the alpha-expression. It is used only in logical expressions. (For exact use of the ALL function, see Chapter 3, Section 3.8.)

Examples:

```
100 LET A$=ALL(B$)
200 C$=AND ALL(D$)
```

AND Logical Operator

```
General Form:

 [LET] Alpha-Receiver = [logical exp] AND logical exp


   logical exp - see Section 3.8
```

Purpose:

The AND operator logically AND's two or more alphanumeric arguments.

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$
```

which logically ANDs A$ and B$ and places the result in A$.

```
100 A$ = B$ AND C$
```

which logically ANDs B$ and C$ and places the result in A$.

Examples:

```
HEX(0F0F) AND HEX(0F0F)=HEX(0F0F)
HEX(00FF) AND HEX(0F0F)=HEX(00FF)
```

See Chapter 3, Section 3.8 for more information on logical expressions.

BIN Function

```
General Form:

 BIN(expression [,d])

        where:  d = 1,2,3,4   (default = 1)
```

Purpose:

This function converts the integer value of the expression to a d-character alphanumeric value which is 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:

```
                                {256      (d=1)
      0 ≤ val. expression <     {65536    (d=2)
                                {16777216 (d=3)
```

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 ≤ 2147483647

Examples:

```
100 A$=BIN(A,4)
200 B$=BIN(A,3) AND BIN(B,3)
```

BOOLh Logical Operator

```
General Form:

 [LET] Alpha-Receiver = [logical exp] BOOLh logical exp

   logical exp - see Section 3.8

          h = a digit from 0 to 9, or a letter from A to F
```

Purpose:

BOOL is a generalized logical operator which 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 II-A). 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 hexdigit following BOOL. A total of 16 logical operations are available (see the table on the next page). The hexdigit 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 OR'ed with 1010, the result is 1110, or hexdigit E. Note that several commonly used BOOL operations are available as separate operators: BOOLE is equivalent to OR, BOOL6 to XOR, and BOOL8 to AND.

Table II-B

BOOL digit	Logical Operation (Note: iff = if and only if)
0	null (bits always = 0; logical inverse of of BOOL F)
1	not OR (1 iff corresponding bits of both arg 1 and arg 2=0)
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)
7	not AND (0 iff corresponding bits of both arg 1 and arg 2=1)
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)

Note: BOOL6 is equivalent to XOR
BOOL8 is equivalent to AND
BOOLE is equivalent to OR

Examples:

```
HEX(0F0F) BOOL1 HEX(0FF0) = HEX(F000)
HEX(0F0F) BOOL5 HEX(0FF0) = HEX(F00F)
HEX(0F0F) BOOLF HEX(0FF0) = HEX(FFFF)
```

CALL

```
General Form:

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

where:  "name" = 1-8 alphanumeric characters; (no embedded
                 spaces)
                 1st must be alphabetic (including @, #, $)
               = SUB "name" of the SUB program being called.

        Note:    Name must be enclosed in quotation marks.

          arg  = {expression
                 {alpha-expression,
                 {array-designator
                 {file-expression
```

CALL directs execution to the named subroutine, identified by a SUB statement, and passes the arguments, if any, 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 or EZBASIC.)

The argument list in the CALL statement must correspond item-for-item with the argument list in the SUB statement, according to Tables II-B and II-C

Table II-C

CALL argument	SUB argument
(alpha-)expression	scalar variable
matrix	matrix
vector	vector
file-expression	file-number

Table II-D

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" (A$, B, I2%(), #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 legal:

```
CALL "HENRY" (STR(C1$()), A(1), B%( ), #N)
```

Note: 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. For non-ADDR type

   - The file-expression is passed to the SUB program to replace the dummy file number. (Specifically, the UFB address is passed to the SUB program.)

   - Arrays and receivers: Current storage addresses are passed to the SUB routine, including pointers to the dimensions and lengths.

   - Other expressions and alpha-expressions: Since these are not receivers, they must be computed and stored in temporary locations, along with their dimensions and lengths.

     Otherwise, execution proceeds as in arrays and receivers, except that returned values and lengths are effectively lost, since the locations are no longer accessible to the calling program.

2. For ADDR-type:

   - 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.

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
```

CLOSE

General Form:	CLOSE file-expression

This statement closes a file which had previously been 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), file, library, and volume need not be respecified by the program or the user.

Attempting to close a file which has not previously been opened by an OPEN statement causes a nonrecoverable program error at run-time.

All files are closed at the start of the program, and opened files should be closed before the end of the program.

Examples:

```
100 CLOSE #1
200 CLOSE #A
300 CLOSE #LEN(A$)
```

```
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 defines scalar variables or arrays which are to be used in common by several program segments.

This statement provides array definition identical to the DIM statement for array variables; the syntax for one COM statement can be a combination of array variables (i.e., A(10), B(3,3)) and scalar variables (i.e., 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 CALL'ed subprograms, the COM statement must be included in the main program and in 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).

Examples:

```
100 COM A(10),B(3,3),C2
200 COM C,D(4,14),E3,F(6),F1(5)
300 COM M1$,M$(2,4),X,Y
400 COM A$10,B$(2,2)32
```

CONVERT

```
General Forms:

    1.  CONVERT alpha-expression TO numeric variable
        [,DATA {GOTO } line number]
               {GOSUB}

                 or

    2.  CONVERT expression TO alpha-receiver, (image)

                           [{#}     ][ [{#}     ]]      {+ }
                           [{0}     ][ [{0}     ]]      {- }
where:  image = [±][$][{*} ... ][.[{*} ... ]][↑↑↑↑][{++}]
                           [{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 also can 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 converting it to numeric.) The alpha-expression may contain blanks anywhere, as with NUM. If, however, the alpha-expression is entirely blank, an error will result (or the data exit will be taken).

Form 2: Numeric-to-Alpha Conversion (Same as PRINTUSING)

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$, (####)
        Result: A$ = "  10"
```

Examples: (Alpha to Numeric)

```
100 CONVERT A$ TO X
200 CONVERT STR(A$,1,NUM(A$)) TO X(1)
```

Examples: (Numeric to Alpha)

```
100 X = 12.195
200 CONVERT X TO A$,(000)
    (result:  A$ = "012")
300 CONVERT X*2 TO A$,(+##.##)
    (result:  A$ = "+24.39")
400 CONVERT X TO STR(A$,3,8),(-#.#↑↑↑↑)
    (result:  STR(A$,3,8) = " 1.2E+01")
500 CONVERT X TO A$, (0000.#####)
    (result:  A$ = "0012.19500")
```

COPY

```
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:

1. The receiver is filled,
   or
2. 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 from the memory location; otherwise, the alpha-expression is constructed in a separate location and copied from there. Thus, COPY'ing a receiver onto itself can result in desirable or undesirable single-character propagation or other position-dependent results.

Examples:

```
100 COPY A$ TO B$
200 COPY - STR(A$(1),3,5) TO C$
```

DATA

```
------------------------------------------------------------------
|                                                                |
| General Form:                                                  |
|                                                                |
|                       [           ]                            |
|     DATA {constant} |,{constant}|                              |
|          {literal } | {literal }|...                           |
|                       [           ]                            |
|________________________________________________________________|
```

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 of the program 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 are entered, 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.

Examples:

```
400 DATA 4,3,5,6,HEX(7A)
500 DATA 6.56E+45, -644.543
```

DATE Function

```
General Form:

                DATE
```

Purpose:

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$,5,2);"/";
300 STR(A$,1,2)
```

Output: 08/10/78

DEFFN

```
General Form:

        DEFFNa[%](v) = expression

    where a = the identifier, a letter or digit which
              identifies the function
          v = the dummy variable, a numeric scalar variable

    If '%' is present, the result is an integer.
```

The "define function" statement, DEFFN, 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 DEFFN is used.

Example:

```
      100 X=3
      200 DEFFN A(Z) = Z↑2-Z
      300 PRINT X + FNA(2*X)
      400 END
Output:  33
```

Processing of FNA(2*X) in the above example proceeds in the following order:

1. Evaluate the FN expression for the scalar variable (i.e., 2*X=6).

2. Find the DEFFN with the matching identifier (i.e., A).

3. Set the dummy variable (line 200) equal to the value of the evaluated expression (i.e., Z=6).

4. Evaluate the DEFFN expression and return the calculated value (i.e., Z↑2-Z).

The above example prints the value 33, since 3 + (6↑2 - 6)= 33.

The DEFFN statement may be entered from any place in a program, but is not executed unless referenced by the program. The expression may be any valid <u>numeric</u> expression. The following restrictions apply:

1. A DEFFN function may not refer to itself; for example,

   ```
   DEFFN A(X) = X + FNA(X)
   ```

   is illegal.

2. Two DEFFN functions may not refer to each other. For example, the following combination of statements is illegal.

   ```
   DEFFNA(A) = FNB(A)
   DEFFNB(A) = FNA(A)
   ```

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 DEFFN statement can have a name identical to that of a variable used elsewhere in the program or in other DEFFN statements; current values of the variables are not affected during FN evaluation. DEFFN statements may also use other variables, whose current values at calling time are used.

A total of 72 user-defined functions may exist in one program (A-Z, 0-9, A%-Z%, 0%-9%).

Examples:

```
600 DEFFN A(C) = (3*A) - 8*C + FNB(2-A)
700 DEFFN B(A) = (3*A) - 9/C
800 DEFFN4(C) = FNB(C)*FNA(2)
```

DEFFN'

```
General Form:

    DEFFN' int {[(receiver[,receiver]...)]}
               {literal [;literal]...      }

    where int = {1 to 32 for program function key
                {                             entries
                {0 to 255 for internal program references
```

The DEFFN' 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.

Repeated subroutine calls executed without RETURN or RETURN CLEAR statements may cause memory overflow. (See RETURN and RETURN CLEAR.)

Keyboard Text Entry Definition

To be used for keyboard entry, the integer in the DEFFN' statement must be a number 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 DEFFN'31 "April Is The Cruelest Month."
200 DEFFN'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 DEFFN' statement, followed by an integer and an optional receiver list enclosed in parentheses, indicates the beginning of a marked subroutine. The subroutine may be entered from the program via a GOSUB' statement (see GOSUB'), 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 DEFFN' 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 depressed or a GOSUB' statement is executed, the execution of the BASIC program transfers to the DEFFN' 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 DEFFN'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.

The DEFFN' 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 (see GOSUB'). 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 DEFFN'2 (A,B(3),C$)
•
•
400 RETURN
```

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 the wrong type of data is given, the entries will be refused, the cursor will be returned to the beginning of the field, and the program will wait for further operator action.

Example:

```
When the previous program is run:
STOP 1.2, 3.24, "JOHN" (now depress PF Key 2)
```

The DEFFN' 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 DEFFN'4
200 INPUT "RATE",R
300 C = 100 * R - 50
400 PRINT "COST=";C
500 RETURN
```

When a DEFFN' 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 DEFFN'1
210 INPUT "ENTER NEW RATE",R
220 RETURN
```

```
Display:  ENTER AMOUNT?
          (Depress PF Key 1)
          ENTER NEW RATE? 7.5
          ENTER AMOUNT?
```

DEFFN' 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

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.

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]

          where:    {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.

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 are defaulted to 10 by 10 matrices.

3. Arrays or variables dimensioned in a COM statement may <u>not</u> be respecified in a DIM. (See COM.)

   A variable or array may occur in <u>only one</u> DIM or COM in each program or subprogram.

Arrays may be redimensioned by using [MAT] REDIM.

Examples:

```
100 DIM A$100
200 DIM A$(4,4),B$(12,12)20,B(3,7)
300 DIM A(10),B$(20)10
```

DIM Function

```
General Form:

    DIM (array-designator, {1})
                           {2}
```

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 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

```
General Form:

  DISPLAY   list     [,list]...

   where:

     list  = { COL (int)                              }
             { AT(exp2, exp3)                         }
             { 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.

Both single values and arrays may be output.

DISPLAY works in generally the same way as ACCEPT, with the following exceptions:

1. Values are <u>written</u> only; no new values are accepted. (Thus there are no PF key clauses or FAC characters.)
2. Pseudoblanks are not used.

Otherwise, see ACCEPT. Note that the screen is cleared prior to DISPLAY, and that a STOP statement should be used in order to halt execution for viewing (if desired) following DISPLAY.

See Chapter 5 for more information on screen I/O.

Examples:

```
100 DISPLAY COL(10),A$,CH(20),AT(20,20),A,PIC(##.##)
200 DISPLAY B$,BELL
```

```
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 executed, 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 Chapters 7 and 8 of the VS Programmer's Introduction for procedures and for the use of return codes.)

FMT

```
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 used to format data values for PRINTUSING and disk I/O statements. It 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 only 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. XX [(int)]
   Skip int positions (input) or write n blanks (output). Omitted int=1.

2. COL (int) or POS (int)

   Next form-spec to start at position int in record or output line.

   (For disk I/O, int ≤ 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. TAB (int)

   Like COL, but all skipped-over characters are set to blank.

4. SKIP [(int)]

   Skip int lines (default=1). Like PRINT SKIP. (Not for disk I/O.)

Data-Spec (Note: w and d are int values.)

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)

```
                [{#}      ][ [{#}      ]]        {+ }
                [{0}      ][ [{0}      ]]        {- }
5.  PIC([±][$][{*} ... ][.[{*} ... ]][↑↑↑↑][{++}])
                [{B}      ][ [{B}      ]]        {--}
                [{/}      ][ [{/}      ]]
                [{,}      ][ [{,}      ]]
```

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

```
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 training  |
|     sign cannot both be specified. |
|                                    |
| 2.  If no signs are present, the   |
|     absolute value of the number   |
|     is printed.                    |
|____________________________________|
```

```
Leading      { +    '+' if > 0, '-' if < 0
             { -    blank if > 0, '-' if < 0
             { $    '$' precedes the number
```

(The above three characters <u>float</u> to the leftmost nonzero digit location.)

Examples:

```
100 FMT PIC(##.##↑↑↑↑)
200 FMT SKIP(10),CH(50),SKIP(-5),COL(20),PIC($**.##)
```

FN

```
General Form:          FN a [%] (expression)

where:  a [%] is a function identifier defined in a DEFFN
        statement
```

FN is used to call a function defined by the DEFFN statement which contains the same identifier. The variable identifiers in the FN expression need not be the same as the dummy variable identifiers in the associated DEFFN statement (see DEFFN).

Examples:

```
100 DEFFN A(A) = 3*A
200 J = FNA (B) + K
```

FOR

```
General Form:

    FOR numeric scalar variable = exp1 TO exp2 [STEP exp]
```

The FOR statement and the NEXT statement 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 (see NEXT). The program lines in the range of the FOR statement are executed repeatedly, beginning with variable = exp1, 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 in the wrong direction, 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 range" (if indeed such a range may be specified), provided that a NEXT without an open FOR is not encountered; this event will cause an error.

> NOTE:
>
> If the loop variable is an <u>integer</u> variable, exp1, exp2 and the step exp will be truncated to integers and all loop calculations will be integer type.

Examples:

```
100 FOR A=1 TO 10 STEP .3
200 PRINT A
300 NEXT A
```

FS Function

```
General Form:

    FS (file expression)
```

FS returns the file status item for the last IO operation on the specified file. Its length is 2 characters. Possible values are:

Character 1	Character 2	Error
0	0	Successful completion of I/O operation.
1	0	End of file reached
2	1	Key out of sequence (OUTPUT mode).
2	2	Duplicate Key
2	3	Record not found (CONSEC files)
		Key not found (INDEXED file)
2	4	Key greater than highest key in file (IND/READ)
		No more room in the file (IND/OUTPUT mode)
3	0	I/O hardware error
3	4	No more room in file (other than 24).
9	5	Invalid function request/sequence.
9	6	Invalid data address.
9	7	Invalid length (WRITE/REWRITE)
9	9	Invalid block format*
		Shared mode I/O errors:
8	0	Invalid Key area (START, READ KEYED)*
8	1	Invalid READ NODATA*
8	2	Label update error*
8	3	Sharing task was terminated*
8	4	Invalid record size/record area* (Record size > 2048)

*not normally encountered by BASIC user

GET

```
General Form:

 GET {file-exp } [ [,] USING line number], arg [,arg]...
     {alpha-exp}

     [, DATA {GOTO } line number]
             {GOSUB}

 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 is that read with the last READ statement; this data is 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.).

Examples:

```
100 GET#A USING 300,B,DATA GOTO 500
300 FMT PIC(####)
```

GOSUB

General Form: GOSUB line number

The GOSUB statement is used to transfer program execution to the first program line of a subroutine. The program line may be any BASIC statement, including a REM statement. 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. The RETURN statement must be the last executable statement on a line, but may be followed by nonexecutable statements as shown below:

```
120 X = 20:GOSUB 200:PRINT X
125
.
.
.
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 information to be accumulated in a table which eventually causes a memory stack overflow error.

GOSUB'

```
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. A subroutine is marked by a DEFFN' statement (see DEFFN'). 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 DEFFN'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 DEFFN' statement (see DEFFN') 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.

Example:

```
100 GOSUB'7
150 END
200 DEFFN'7:SELECT PRINTER (80)
210 RETURN
```

Example:

```
100 GOSUB'12 ("JOHN",12.4,3*X+Y)
200 END
300 DEFFN'12(A$,B,C(2))
400 PRINT A$,B,C(2)
500 RETURN
```

GOTO

General Form: GOTO line number

This statement transfers execution to the specified line number; execution continues at the specified line.

Example:

```
100 J=25
200 K=15
300 GOTO 700
400 Z=J+K+L+M
500 PRINT Z,Z/4
600 END
700 L=80
800 M=16
900 GOTO 400
```

Output:
```
136        34
```

HEX Literal String

```
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.

Examples:

```
100 A$=HEX(0C0A0A)
200 IF A$ > HEX(7F) THEN 100
300 PRINT HEX(8001);"TITLE"
```

HEXPACK

```
General Form:

 HEXPACK alpha-receiver FROM alpha-expression

      [, DATA {GOTO } line number]
              {GOSUB}
```

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	1000
9	1001
A or :	1010
B or ;	1011
C or <	1100
D or =	1101
E or >	1110
F or ?	1111

If the alpha-expression (actual length) 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 or, 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
```

Examples of Valid Syntax:

```
HEXPACK A$ FROM B$
HEXPACK STR(A$,1,3) FROM STR(B$,7)
HEXPACK A$() FROM B$()
HEXPACK A$ FROM "3AFC282C"
```

HEXUNPACK

```
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 effectively 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
```

Examples of Valid Syntax:

```
HEXUNPACK A$ TO B$
HEXUNPACK STR(A$, 5) TO STR(B$, 1, 4)
HEXUNPACK A$() TO B$()
```

IF...THEN...ELSE

```
General Form:

                                         [                         ]
    IF relation THEN {line number}       |ELSE  {line number}      |
                     {executable }       |      {executable }      |
                     { statement*}       |      { statement*}      |
                                         [                         ]
                                              *except another IF

where relation = {alpha exp operator alpha exp     }
                 {                                 }
                 {expression operator expression   }
                 {                                 }
                 {              {AND}              }
                 {    relation  {OR }  relation    }
                 {              {XOR}              }
                 {                                 }
                 {NOT relation                     }
                 {(relation)                       }

                              {< }
                              {<=}
 where 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" is specified, execution continues at the specified line number.

   - 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 <u>false</u>:

- 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. Otherwise, left-to-right execution

> **NOTE:**
>
> Nested IF statements are not allowed.

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
```

Image(%)

```
 General Form:
                 %    {character string    }  ...
                      {format specification}

 where:

         Character string = {any character     }  ...
                            {except '#'        }

         Format specification =

 [    ]                                         [{+ }]
 |{+}|[$]#...[,][#...[,]]...[.][#...][↑↑↑↑]|{- }|
 |{-}|                                          |{++}|
 [    ]                                         [{--}]
```

IMAGE (%) is 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:

Editing Characters

```
Leading { +   '+' if >0, '-' if <0
        { -   blank if >0, '-' if <0
        { $   '$' precedes the number
```

(The above three characters <u>float</u> to just before the leftmost nonzero digit location.)

- `#` digit position - <u>blank</u> 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 <u>significant</u> 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
```

```
 ------------------------------------------
|                                          |
|   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.

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
600 DEFFN' 16
```

INIT

```
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$,B$
```

sets both alpha variables A$ and B$ to contain all question mark characters.

INPUT

```
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, he enters, for example,

```
400 INPUT A,B
        or
400 INPUT "VALUE OF A,B",A,B
```

before the first program line which requires either of these values (A,B). When the system encounters this INPUT statement, it outputs the input 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. Note that 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.

e.g.

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, the literal string need not be enclosed in quotes. However, leading blanks are ignored and commas act as string terminators. (This applies to subroutine parameters also - see PF key section.)

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 DEFFN') and an INPUT statement is executed, pressing the PF key causes the character string in the DEFFN' 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 touched.

For example:

```
100 DEFFN'01"COLOR T.V."
200 INPUT A$
```

Output: ?

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 DEFFN') and the system is awaiting input, pressing the PF key will cause the specified subroutine to be executed. Specifically: 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 beeps if there are too few values or if they do not correspond correctly to the receivers in the GOSUB' argument list. An illegal PF key also causes a beep. When the subroutine RETURN is encountered, a branch will be made back 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 Key 02 is depressed, 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
Run Program
```

Output:

```
AMOUNT? 7  (ENTER)
AMOUNT? 5  (ENTER)
AMOUNT? 11 (ENTER)
AMOUNT? (Depress PF 2)
TOTAL = 23
AMOUNT?
```

KEY

```
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 pseudo-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 give 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

```
General Form:

    [LET] numeric variable [,numeric variable]... = expression
 or
    [LET] alpha-receiver [,alpha-receiver]... =  alpha-expression
 or
    [LET] alpha-receiver = logical expression
```

The LET statement directs the system to evaluate the expression following the equal sign and to assign 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, then only one receiver may appear on the left.

The word LET is optional. If it is omitted, its purpose is assumed.

An error results if a numeric value is assigned to an alphanumeric receiver or if an alphanumeric value is assigned to a numeric variable.

Example 1:

```
400 LET X(3),Z,Y=P+15/2+SIN(P-2.0)
```

Example 2:

```
500 LET J=3
```

Example 3:

```
(In this example, LET is assumed)
100 X=A*E-Z*Y
200 A$=B$
300 C$,D$(2)="ABCDE"
```

Example 4:

```
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. Note that assignment is <u>right-to-left</u>.

> NOTE:
>
> Logical expressions are described in Chapter 3, Section 3.8.

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 READ KEYED, by those alternate key paths.

MAT + (MAT addition)

```
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;
```

```
        [           ]           [           ]
        |1    1    1|           |3    3    3|
Let D=  |1    1    1|     E=    |3    3    3|
        |2    2    2|           |3    3    3|
        [           ]           [           ]
```

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

```
General Form:

    MAT numeric array name1 = {ASORT} (numeric array name2)
                              {DSORT}

    MAT alpha array name1 =   {ASORT} (alpha array name2)
                              {DSORT}
```

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]	(nm)[L]
(n)[L]	(pxq)[k]	(nx1)[L]
(n)[L]	(p)[k]	(n)[L]

An error occurs if array 1 as originally dimensioned 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 <u>not</u> be the same, i.e., sort-in-place is not supported.

> NOTE:
>
> Alphanumeric sorting uses the usual ASCII collating sequence.

Examples:

```
100 MAT A=ASORT(B)
200 MAT A$=DSORT(B$)
300 MAT C$=ASORT(B$)
```

MAT CON (MAT CONstant)

```
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)

```
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.

Examples showing statement syntax:

```
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)

```
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.

Example showing statement syntax:

```
100 MAT A = IDN(4,4)
200 MAT B = IDN
300 MAT C = IDN(X,Y)
```

Example in which the identity matrix is displayed:

```
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

```
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. When the system encounters a MAT INPUT statement, it displays the literal, if given, and a question mark (?) and waits for the user to supply values for the arrays specified in the MAT INPUT statement. 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 which are input 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 in it 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 are 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 are 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 1, 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 on the keyboard the values, separated by commas,

```
-3, -5, .612, .41
```

Key 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; key 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 2, 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.

MAT INV (MAT inverse)

General Form:	MAT c = INV(a)[,d]
	where c and a are 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 <u>singular</u> (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 your 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).

Example 1, illustration of statement syntax:

```
100 MAT A=INV(B)
200 MAT Z1=INV(P),X2
300 MAT F=INV(C),J3
400 MAT C=INV(C)
```

Example 2:

The following program takes the 4x4 matrix A from the keyboard input, calculates the inverse of it, 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

If the input matrix is singular (i.e., non-invertible), D=0.

MAT * (MAT multiplication)

```
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.

Example of statement syntax:

```
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
```

Example of usage in a program:

```
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

```
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 carriage return/line feed (CR/LF) occurs before <u>each</u> row (except the first) is printed, and the last line of the matrix is followed by a CR/LF. A multiple MAT PRINT is treated like several single MAT PRINT's. 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.

Examples of statement syntax:

```
100 DIM A(4),B(2,4),B$(10),C$(6)
200 MAT PRINT A;B,C$
300 MAT PRINT A,B$
```

Examples of usage in a program:

This program takes as input nine alphanumeric quantities, 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$;
```

MAT READ

```
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 system uses the next sequential DATA statement. If no more DATA statements are in the program, an error occurs and execution is terminated.

Alphanumeric string variable 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; if no length is specified, a default of 16 is used.

Example 1:

```
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
```

Example 2:

```
100 DIM A(2,2),B$(3,2)
200 DIM C(3),D$(4)7
300 MAT READ A,B$,C(2),D$(4)6
400 DATA 1,2,3,-3.4E12
500 DATA "ABC","DEFG","HI","J","K","L"
600 DATA .2345,1E-12,"AB","CD","EFGH","IJK"
700 MAT PRINT A,B$,C,D$
```

MAT REDIM

```
General Form:

        MAT REDIM redim-elt[,redim-elt]...

where:  redim-elt = {numeric array name (exp1[,exp2])
                    {alpha array name (exp1[,exp2])[exp3]

where:  {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 REDIM'd.

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, and vice-versa.

Examples:

```
100 MAT REDIM A(10),B$(10,20)10
200 MAT REDIM A(20,30)
```

MAT()* (MAT scalar multiplication)

```
 ______________________________________________________________
|                                                              |
| 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.

Example of statement usage:

```
100 MAT C = (SIN(X))*A
200 MAT D = (X+Y*2)*A
300 MAT A = (5)*A
```

Example of program:

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)

```
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 2 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.

Example of statement syntax:

```
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
```

Example of program:

```
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 you let 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)

```
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.

Example of statement syntax:

```
100 MAT C = TRN(A)
```

Example of program usage:

```
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)

```
General Form:   MAT c = ZER [(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 equal to zero. Using (d1, d2) causes the matrix to be redimensioned. If (d1,d2) are not used, the matrix retains the dimensions specified in a previous COM, DIM, or MAT statement.

Example:

```
100 MAT C = ZER(5,2)
200 MAT B = ZER
300 MAT A = ZER(F,T+2)
400 MAT D = ZER(20)
```

Mathematical Functions

The following General Form 1 applies to most mathematical functions. The general forms for the remainder are listed below.

```
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 trigorometric functions can be easily expressed using these functions in expressions.

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)	same; ATN is a synonym for ARCTAN.

Other Numerical Functions

The remaining twelve 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)
```

MOD	MOD(X,Y)	The modulus function; the remainder of the division of the first element by the second.

```
General Form 4:

    #PI
```

#PI	#PI	The value 3.14159265358979323.

See Section 2.6 for more information on Numeric Functions.

NEXT

```
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. When the NEXT statement is executed, the STEP value 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. ('%' is not a legal numeric character.) A numeric character is defined to be one of the following: digits 0 through 9, and special characters E, . (decimal point), +, -, space. Numeric characters are counted starting with the first character of the specified variable or STR function. 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. Note: If A$ = "1E88", then NUM(A$)=16 even though 1E88 is an illegal value, since it 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 <u>integer</u>.

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

```
General Form:

    ON expression {GOTO } entry [,entry]...
                  {GOSUB}

where:  entry = {line number
                {null

    The last entry must be a line number (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, the transfer is made to that line.
2. If null, no transfer is made.
3. If I<1 or > number of entries, no transfer is made.

In 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

```
General Form:

                                                    {INPUT
                                                    {IO   }
    OPEN [,][{NODISPLAY}[,]] file-exp[,]            {SHARED}
             {NOGETPARM}                            {EXTEND}
                                                    {OUTPUT}


     [,SPACE = exp1] [,DPACK = exp2] [,IPACK = exp3]


     [,FILE = alpha-exp1] [,LIBRARY = alpha-exp2]


     [,VOLUME = alpha-exp3] [,BLOCKS = exp4]

    where:

     alpha-exp1,2,3 = file, library, and volume names
                      must be alphabetic or numeric,
                      enclosed in quotation marks,
                      1st character alphabetic.

         Filename = at most 8 characters (remainder ignored)
         Library  = at most 8 characters (remainder ignored)
         Volume   = at most 6 characters (remainder ignored)

     BLOCKS    = 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 file or create a new file. 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 OPEN'ed 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 II-1.

Attempting to OPEN a file that has already been OPEN'ed and not yet CLOSE'd causes a nonrecoverable error at run-time.

Use of the SPACE, DPACK, IPACK, NODISPLAY, and NOGETPARM fields is explained below.

NODISPLAY, NOGETPARM

When OPEN'ing a file in the program, OPEN will normally issue a GETPARM (see the discussion of the GETPARM SVC in the 2200VS 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 2200VS Programmer's Introduction, Chapter 2.)

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 2200VS Programmer's Introduction.)

The remaining parameters differ in usage depending on whether the file is being OPEN'ed 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

Further use of FILE, LIBRARY, VOLUME:

In any mode, if FILE/LIBRARY/VOLUME are alpha-receivers, the actual names will be returned to the receivers after OPEN.

Table II-E.

Legal Function Requests and Descriptions

TYPE MODE	CONSEC	VAR CONSEC	{ INDEXED } { VAR INDEXED }	TAPE	PRINTER
	Ops: READ,SKIP	Ops: READ,SKIP	Ops: READ	Ops: READ,SKIP	NOT ALLOWED
INPUT (old files only)	Consecutive or relative READ or SKIP, starting from beginning of the file.	Consecutive or relative READ or SKIP starting from beginning of file.	Consecutive or keyed READ, starting from beginning or after last record read.	Consecutive or relative READ or SKIP starting from beginning of file.	
	Ops: READ,REWRITE, SKIP	Ops: READ,SKIP, REWRITE	Ops: READ,WRITE, REWRITE,DELETE	Ops: READ,SKIP	NOT ALLOWED
IO (old files only)	Consecutive or relative READ or SKIP from beginning of file, with HOLD/ REWRITE option.	Consecutive or relative READ or SKIP from beginning of file, with HOLD/REWRITE option	Consecutive or keyed READ or WRITE from beginning of (key) file, with HOLD/ REWRITE/DELETE option.	Consecutive or relative READ or SKIP from the beginning of the file with HOLD option	
	NOT ALLOWED	Ops: WRITE, HOLD, RELEASE	Ops: READ,WRITE, REWRITE,DELETE HOLD, RELEASE	NOT ALLOWED	NOT ALLOWED
SHARED (old INDEXED files; old or new CONSEC files)		Used for (variable length) log files.	Same as IO, but allows multiple access, independently, HOLD protection.		
	Ops: WRITE	Ops: WRITE	Ops: WRITE	Ops: WRITE	Ops: WRITE
OUTPUT (new files only) (old files deleted)	Writes records consecutively to a new file.	Writes records consecutively to a new file.	Writes records to a new file - (primary) keys must be in ascending order.*	Writes records consecutively to a new file.	Writes records consecutively to a new file.
	Ops: WRITE	Ops: WRITE	NOT ALLOWED	NOT ALLOWED	NOT ALLOWED
EXTEND (old files only)	Writes records consecutively, starting at the current file end	Writes records consecutively, starting at the current file end			

*To write records in Random Key order, do the following:

1. OPEN OUTPUT
2. CLOSE
3. OPEN IO
4. WRITE RECORDS

Examples:

```
100 OPEN NODISPLAY #1,IO,FILE="THOMAS",LIBRARY="STEARNS",!
200 VOLUME="ELIOT"
300 OPEN #2,OUTPUT
```

OR Logical Operator

```
General Form:

 [LET] Alpha-Receiver = [logical exp] OR logical exp


   logical exp - see Section 3.8
```

Purpose:

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 0100 0001 in binary; a blank space is HEX(20) or 0010 0000 in binary. When two characters are OR'd, a binary one in either becomes a binary one in the result. Thus, OR'ing "A" with " " produces binary 0110 0001 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 Chapter 3, Section 3.8 for more information on logical expressions.

POS Function

```
General Form:

                                      {< }
                                      {<=}
                                      {> }
        POS ([-] alpha-expression {>=}  alpha-exp)
                                      {<>}
                                      {= }
```

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_)

Examples:

```
100 A%=POS(-A$<STR(B$,2,2))
200 FOR A=1 TO 10 STEP POS(C$=B$)
```

PRINT

```
General Form:

        PRINT [prt-elt][t [prt-elt]]...[t]

where:  t = comma or semicolon

  prt-elt = {character prt-elt
            {control prt-elt

  character prt-elt = {expression               }
                      {alpha-expression         }
                      {HEXOF (alpha-expression)}

    control prt-elt = {BELL                     }
                      {PAGE                     }
                      {SKIP[(exp )]             }
                      {TAB (exp )               }
                      { COL(exp)                }
                      {AT( exp , exp [,[exp]]) }
```

The PRINT statement routes output to the currently selected PRINT device (printer or workstation) - see SELECT statement.

For PRINT output, the output line is divided into as many _zones_ of 18 characters as possible; thus the printer has seven zones and the workstation has four. Note that the last zone (maximum) may be longer than the rest, extending to the end of the line.

PRINT executes as follows:

1. Character print elements are printed starting at the next unused print position and ending at the last used print position. If too long, the prt-elt is continued from the beginning of the next line(s).

2. Control print elements are output based on the current print position, and end at the new position.

3. A comma anywhere except immediately after a control prt-elt moves the print position from its current zone to the start of the next available zone (if in zone 7, moves to column 1 of the next line). A comma after a control prt-elt is treated like a semicolon.

4. A semicolon causes no change in print position. Note that at least one comma or semicolon is required between any two prt-elts.

5. A PRINT ending in a comma, semicolon, or control prt-elt causes the cursor to remain at the next available print position; any other PRINT (including an empty PRINT) is followed by a CR/LF.

Note that 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.

Print Elements

1. expression

- If $|exp|<10^{-1}$ or $|exp|\geq 10^{15}$, 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 746.79;
         b7.4679000000E+02b
       start              end
  ```

- If $10^{-1} \leq |exp| < 10^{15}$, the format is fixed-point:

  S[Z...][DF...]b

  where:
  - S = minus sign if negative, blank otherwise
  - Z = digit
  - D = decimal point
  - F = digit
  - b = trailing blank

  and total Z's + total F's ≤ 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. (Note that this includes trailing blanks - HEX(20)).

```
    e.g.    100 A$ = "ABC"
            200 PRINT "HEX VALUE OF A$ =";HEXOF (A$)
Result: HEX VALUE OF A$ = 41424320202020202020202020202020
```

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: Beeps the CRT; 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. 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. TAB(n)

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 (HEX(20)) passed-over characters. If the column has been passed the TAB is ignored.

In either case, if the tab position is greater than the SELECT'ed line length, the print position is advanced to column 1 of the next line. If N is negative or zero, the TAB is ignored.

```
100 LET   A = 3

200 PRINT TAB(30),
```
A will be printed in the 30th column.

8. COL(n)

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. AT(r, c[, [e]])

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:

- $1 \le r \le 24$.
- $1 \le c \le 80$.
- $e \ge 0$; if too large, only characters to the end of the screen are erased.

  If e and 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 no effect on passed-over characters.

PRINTUSING

```
General Form:

    PRINTUSING line number [,prt-elt[{,}prt-elt]...][;]
                                      {;}

where prt elt = {expression
                {alpha-expression

    line number = line number of IMAGE or FMT.
```

PRINTUSING routes formatted output to the currently selected PRINT device, using the referenced IMAGE or FMT statement.

PRINTUSING starts at the current print position and ends with the cursor either at the next print position (if final ';' present) or at the beginning of the next line (if no final ';').

The IMAGE or FMT may be reused if there are more elts than format-specs. If the delimiter following the last displayed value is a comma, the cursor moves to the beginning of the next line before re-use. if the delimiter is a semicolon, the cursor is not moved.

Example:

```
100 PRINTUSING 200, N
200 % ##.##↑↑↑↑
```

```
General Form:

 PUT {file-exp       } [[,]USING line number],arg[,arg]...
     {alpha-receiver}

 [,DATA {GOTO }  line number]
        {GOSUB}

 where:  arg = {expression
               {alpha-expression
               {array-designator
```

PUT inserts data into the referenced record area or alpha-receiver USING the referenced Image or FMT, if specified, or using standard format.

PUT does <u>not</u> 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.

EXAMPLE:

```
SELECT #1 "EXAMPLE" CONSEC, RECSIZE=16
OPEN #1 EXTEND, FILE="EXAMPLE", LIBRARY="DATA",VOLUME="VOL444"
PUT#1,B$
```

READ

```
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 statements and DATA statements must be used together. If a READ statement is referenced beyond the limit of values in a DATA statement, 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.

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 Disk File

```
General Form:
                                  [                                   ]
                                  |  {            {> }              }|
    READ file-exp [[,]HOLD]|[,]{KEY[exp1]{>=}alpha-exp1}|
                                  |  {            {= }              }|
                                  |  {RECORD=exp2                   }|
                                  [                                   ]

    [[[,]USING line number],arg[,arg]...]

    [,EOD{GOTO }line number] [,DATA{GOTO }line number]
         {GOSUB}                        {GOSUB}

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.

        exp1 =  Alternate key number for keyed READ on
                alternate indexed file (PRIMARY key used if
                exp1 = 0 or omitted).

  alpha-exp1 =  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 of FMT or Image statement
                describing the input data format.

         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 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 "Key of Reference" in the case of ALTERNATE INDEXED files, i.e., the last used in READ KEY).

If no arg list is present, the data is left unconverted in the buffer, and is accessible only through GET.

If USING is omitted, data is assumed to be in internal format.

Example:

```
SELECT #1 "EXAMPLE" CONSEC, RECSIZE=16
OPEN #1 INPUT, FILE="EXAMPLE",LIBRARY="DATA" VOLUME="VOLUME"
READ #1,B$
```

REM

```
----------------------------------------------------------------------
|                                                                    |
| General Form:               REM [text string]                      |
| where:  text string = any characters or blanks (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 system 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.

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.

RESTORE

```
General Form:

      RESTORE {[expression]
              {LINE = line number[, expression]}

where:

     line number = line number of a DATA statement 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)th 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.

If 'expression' is omitted, the pointer is set to the first data value in the program or in (or beyond) the specified DATA statement.

Examples of valid syntax:

```
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

```
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. (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 PRINTUSING 500,X,N$
500 % COST = $#,###,###.## CODE = ####
600 RETURN
```

RETURN CLEAR

General Form: RETURN CLEAR [ALL]

Clears subroutine return-address information, generated by the last or all executed subroutine calls, from memory.

The RETURN CLEAR statement is a dummy RETURN statement. With the RETURN CLEAR statement, subroutine return address information from the last previously executed subroutine call is 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. Thus, no RETURN or RETURN CLEAR may be executed before a subsequent GOSUB or GOSUB'.

The RETURN CLEAR statement is used to avoid memory 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.

Example:

```
100 DEFFN'15
200 RETURN CLEAR
```

NOTE:
If a program repeatedly exits from a subroutine without executing a RETURN or RETURN CLEAR statement, an error may result.

REWRITE

```
General Form:

    REWRITE file-exp [[,]SIZE = exp][[,]MASK = alpha-exp1]
          [[[,]USING line number],arg[,arg]...]

        [,DATA {GOTO } line number]
               {GOSUB}

where:

    USING line# = line number of FMT or Image describing
                  the output format.

        arg = {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, if specified. Otherwise, standard format is used.

If the file is <u>not</u> 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 <u>ignored</u>.

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 <u>not allowed</u> for <u>CONSEC VARC</u> files.

MASK is used to set the alternate Key mask for alternate indexed files. See the explanation of the MASK system 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

General Form:

```
RND(exp)
```

Purpose:

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 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 described above. 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

```
General Form:   ROTATE [C] (alpha-receiver, expression)

where -8 ≤ expression ≤ 8
```

This statement rotates bits in the given alpha-receiver. If exp < 0, rotation is left to right |exp| bits. If exp > 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.

```
e.g.,   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

```
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 right 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(123.456,7-2) = 100.0000
ROUND(123.4567,-3) =    0   etc.
```

Note that "rounding upward" occurs, unlike the INT function; if ROUND is told to round 4.7 to 0 decimal places, it will produce 5, not 4.

SEARCH

```
General Form:
                              {< }
                              {<=}
   SEARCH [-] alpha-exp1 {> } alpha-exp2
                              {>=}
                              {<>}
                              {= }


         TO {numeric array-designator}  [STEP expression]
             alpha-receiver
```

SEARCH searches alpha-exp1 (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-exp1: 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-exp1 minus actual length alpha-exp2 +1)th byte of alpha-exp1. 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 exp1 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 <u>pair</u> of bytes will contain the 2-byte binary representation of the starting positions, as with (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 <u>lost</u>.

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:

```
0013000D0008000400030001000002020
 1        3         4         8
 13       19        0         0
```

SELECT

```
General Form:

    SELECT select-elt [,select-elt][...]

where:  select-elt = {P[d]                                    }
                     {R                                       }
                     {D                                       }
                     {G                                       }
                     {PRINTER [(exp)]                         }
                     {CRT                                     }
                     {POOL file number[,file number]...,BLOCKS=int}

where P[d] = d/10 second execution pause after each write to the work-
             station.  If d=0 or omitted, no pause.  System default = no
             pause.

        R}   trig arguments/results in radians, degrees or grads,
        D} = respectively.  (360 degrees = 2  radians = 400 grads.)
        G}   System default = radians

  PRINTER}   = route PRINT'ed output (PRINT, PRINTUSING, etc.) to the
  CRT    }     line printer or workstation, as specified.  If no SELECT
               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 ≤ 132

             (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 P(9),PRINTER,D,POOL#1,#2,BLOCKS=2
```

SELECT File

```
SELECT file-number [,] "prname"[,] { Consecutive }  [,IOERR exit]
                                    { Indexed}
                                    { Tape  }
                                    { Printer}

File-number = #n, where n is an integer from 1 to 64

prname = 1-8 characters (alphanumeric, including @,#,$; first must be
         alphabetic or @,#,$)

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

tape = [VAR[C][,]] TAPE, NL, RECSIZE = int7, BLKSIZE = int8,

       DENSITY = { 800}[,EOD exit]
                 {1600}

printer = PRINTER, RECSIZE = int10

exit = {GOTO} line number
       {GOSUB}
```

Purpose:

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 - a file may be read from or written to a tape. Only the first file on the tape can be read without changing the file sequence number given in the OPEN GETPARM, and only non-labeled tapes can be used. READ, WRITE, GET, PUT, and SKIP may be used.

- Printer - files may be written for use by the printer. The first two bytes in each record must be printer control characters (see <u>2200VS Principles of Operation</u>). 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 SELECT's must appear before any file I/O statements in the program.

file-number	pound-sign (#) followed by an integer from 1 to 32 (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: the first must be alphabetic (alphabetic includes $, #, 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≤int1≤2048}
{                             }
{VAR CONSEC        1≤int1≤2024}
{                             }
{INDEXED           1≤int1≤2040}
{                             }
{VAR INDEXED       1≤int1≤2024}
```

KEPOS = 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 disk 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 1 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.

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=15BLKSIZE=1000     !
700 DENSITY=1600,EOD GOSUB 1000

800 SELECT#5,"WHERE",PRINTER,RECSIZE=134
```

SIZE Function

```
General Form:

    SIZE (file expression)
```

SIZE returns the size of the last record read from the specified file. The result is an integer.

SKIP

```
General Form:

    SKIP file-exp{[,]BEG} [,EOD{GOTO } line number]
                 {,exp  }       {GOSUB}

where:

    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 re-read by the next READ or GET statement.

A SKIP value of 0 is effectively ignored.

Examples:

```
100 SKIP #A,BEG
200 SKIP #1,B,EOD GOTO 1000
```

STOP

```
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-executed.

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.

Examples:

```
100 STOP A$
200 STOP
300 STOP"THE SUN BEATES, AND THE DEAD TREE GIVES NO SHELTER"
```

STR Function

```
General Form:

                STR( {alpha expression  } [, s [,n]])
                     {alpha array string}
    where s   =  starting character in sub-string (an
                 expression) (1 if omitted)
          n   =  number of consecutive characters desired
                 (an expression)
          s      cannot be zero or negative
          n      cannot be zero or negative
```

Purpose:

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,

```
100 B$ = STR(A$,3,4)
```

sets 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. For example,

```
100 A$ = "ABCDE"
200 PRINT STR(A$,3)
```

produces CDE at execution time.

Examples of Syntax:

STR(A$,3,4)	Takes the third, fourth, fifth and sixth characters of A$.
STR(A$,3)	Starting with the third character in A$, takes the remaining characters of A$.
STR("THE CRICKET NO RELIEF",13,2)	Returns NO.

Example:

100 DIM A$20	
200 B$ = "ABCDEFGH"	Assigns the value ABCDEFGH to B$.
300 A$=STR(B$,2,4)	Assigns the value BCDE to A$.
400 STR(A$,4)=B$	Assigns the value ABCDEFGH to characters 4 through 11 of A$.
500 STR(A$,3,3)=STR(B$,5,3)	Assigns the value EFG to the third, fourth and fifth characters of A$.
600 IF STR(B$,3,2)="AB" THEN 100	Compares the third and fourth characters of B$ (CD) to the literal AB.
700 READ STR(A$,9,9)	Assigns the next data value read to characters 9 through 17 of A$.
800 DATA "A1B2C3D4E5F6G7H819"	

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. For example,

```
      100 A$ = "123456789"
      200 STR(A$,3,5) = "ABC"
      300 PRINT A$
output at execution time:     12ABC  89
```

Examples:

```
100 A$ = STR(B$,2,4)
200 STR(D1$,I,J) = B$
300 IF STR(A$,3,5)>STR(B$,3,5) THEN 100
400 READ STR(A$,9,9)
500 PRINT STR(C$,3)
600 DIM L$(5)
700 LET M$ = STR(L$(),3,20)
```

SUB

```
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. 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.

SUB "name" 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 subprogram address.

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 the description of argument passing in Chapter 4.

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:

    1. 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.

    2. 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 SUB routine as in (A). Although values may be changed in the SUB routine, 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 is passed pointers to the _Locations_ of the passed arguments. All _array dimensions_ and alphanumeric _Lengths_ are as specified _in the SUB program_ (or 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. If the data is in a user-accessible (non-temporary) location, he may access the changed values.

  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.

SUBroutines may call other SUBroutines, but may not be called recursively.

A source file may contain exactly one module, either a program or a SUBroutine.

Examples:

```
500 SUB"AND"
600 A$=STR("THE DRY STONE NO",5,3)
700 PRINT A$;"SOUND OF WATER"
800 END

100 SUB"ONLY"ADDR(A$,B,B(),#N_
200 IF A$ AND "THERE IS A SHADOW" THEN B=20
300 END
```

TIME Function

```
General Form:

   TIME
```

Purpose:

TIME returns an 8-character string containing the current time (accurate to hundreths of a second) in the form HHMMSShh (24-hour clock-midnight is 0:00; 3 PM is 15:00, and so forth). The TIME function takes no argument.

Example:

```
100 PRINT "THE TIME IS ";STR(TIME,1,2);":";STR(TIME,3,2);":";     !
200 STR(TIME,5,2); " O'CLOCK"
300 PRINT SKIP(-1)
400 GOTO 100
```

TRACE

```
General Form:        TRACE [OFF]
```

The TRACE statement provides debugging information useful for "tracking" the execution of a BASIC program. TRACE mode is turned on in a program when a TRACE statement is executed and turned off when a TRACE OFF statement is executed. When the TRACE mode is on, output on the printer is produced when:

1. Any program receiver is assigned a new value during execution (LET, READ, FOR statements, etc.).

   format: receiver = received value

2. A program transfer is made to another sequence of statements (GOTO, GOSUB, GOSUB', IF, NEXT).

   format: TRANSFER TO line number

Example 1:

```
 50 TRACE
100 DIM Z(5)
200 A,B,C=2
300 LET X,Y,Z(5)=A+SIN(B)/C
```

Output:

```
A=2
B=2
C=2
X=2.4546487134284
Y=2.4546487134284
Z() = 2.4546487134284
```

Example 2:

```
300 TRACE
400 READ A,B,C,D
500 DATA 9.4, 64.27, 137492.1E8, 99.4
```

Output:

```
A=9.4
B=64.27
C=137492100E+13
D=99.4
```

Example 3:

```
100 GOTO 200
```

Output:

```
TRANSFER TO 200
```

Example 4:

```
300 GOSUB 10
```

Output:

```
TRANSFER TO 10
```

Example 5:

```
 50 TRACE
100 DIM X(3)
200 FOR I=1 TO 3
300 PRINT X(I);
400 NEXT I
```

Output:

```
I=1
0
I=2
TRANSFER TO 200
0
I=3
TRANSFER TO 200
0
I=> (end-of-loop indicator)
```

Example 6:

```
100 A$=HEX(414243)
```

Output:

```
A$="ABC"
```

Example 7:

```
100 DIM A(4)
200 A(1)=24.2:A(2)=25.36:A(3)=48.001:A(4)=14.759
300 FOR I=1 TO 4
400 TRACE
500 X = X+A(I)
600 TRACE OFF
700 NEXT I
```

Output:

```
X = 24.2
X = 49.56
X = 97.561
X = 112.32
```

TRAN

```
General Form:

      TRAN (alpha-receiver, alpha-expression) [R]
```

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:

   - R 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. If so, the subject byte is changed to the corresponding "translate-to" character. If a matching byte is not found, the subject byte is not changed.

   - R not specified: The alpha-expression is treated as a table of consecutive "translate to" bytes. The subject byte is changed to the (n+1)st 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$, "MJAORHYN")R
400 TRAN (B$, "ABCDEF")
500 PRINT A$,B$
```

produces

```
MARY            ABCD
```

VAL

```
General Form:   VAL(alpha-exp[,d])

where:  d = 1,2,3,4 (default = 1)
```

This function converts the first d characters of the specified alphanumeric value to an integer. 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.

Examples of Syntax:

```
100 X = VAL(A$)
200 PRINT VAL("A")
300 IF VAL(STR(A$,3,1))<80 then 100
400 Z = VAL(A$)*10-Y
```

(See BIN in Chapter 3 for the formats for various values of d.) Note that VAL returns an integer.

WRITE

```
General Form:

WRITE file-exp[[,]SIZE=exp][[,] MASK = alpha-exp ]
                                                1
     [[[,]USING Line number],arg[,arg]...]

    [,EOD{GOTO }line number][,DATA{GOTO }line number]
         {GOSUB}                  {GOSUB}

where:

        SIZE =  record size for VAR files

        MASK = 2 byte mask for alternate indexed files.  (If
               only 1 byte, right-padded with HEX(00))

 USING line# =  line number of Image or FMT describing the
                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).

If an arg list is present, the data is moved one value at a time, using the Image, FMT, or internal formatting. If an arg list is not present, the data is taken directly from the buffer, where it has already (presumably) been formatted with a PUT statement.

For non-VAR[C] files, the record size is as specified in OPEN; 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 6.6). If omitted, the current MASK is used.

Example:

```
100 WRITE#N,SIZE=100,MASK=A$,EOD GOTO 1000, DATA GOTO 1200
```

XOR

```
General Form:

 [LET] Alpha-Receiver = [logical exp] XOR logical exp

   logical exp - see Section 3.8
```

Purpose:

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 Chapter 3, Section 3.8, for more information on logical expressions.

Examples:

```
HEX(0F0F)XOR HEX(0F0F)=HEX(0000)
HEX(00FF)XOR HEX(0F0F)=HEX(F0F0)
```

APPENDIX A

GLOSSARY

alpha: Short for "alphanumeric". One of the three BASIC data types; capable of containing a sequence of characters from the ASCII character set.

argument: An item provided as input to a function or subroutine. The type of argument generally must conform to rules set by the called function or subroutine .

array: A group of variables referred to by the same name, with one or two numeric expressions (subscripts) selecting particular variables (elements). See Section 2.3.

array-designator: An item in the BASIC language which stands for all elements of an array, taken in subscript or row-by-row order. Denoted by the array name followed by opening and closing parentheses, but with no subscripts inside the parentheses, such as: E(), M9$(), W%().

array string: A string formed by concatenating the defined lengths of all the elements of an alpha array. Like array-designators, array strings are denoted by the array name followed by opening and closing parentheses, but with no subscripts, such as: M9$(), W$(), A$(). See Section 3.6.

ASCII: The American Standard Code for Information Interchange, which pairs numbers to displayable characters. The ASCII code is used to represent string data in VS BASIC where applicable.

Boolean: A function which operates on a value on the basis of its individual bits. The AND, BOOL, OR, and XOR statements discussed in Chapter 3 perform Boolean functions.

character: A number, letter, or symbol: the elementary unit of information in the BASIC "alpha" data type.

compiler: The utility program which takes as input a source file containing BASIC statements, and may produce as output an object file executable by the VS processor. The operation of the BASIC Compiler is discussed in Section 1.5.

concatenate: Combine two (or more) BASIC alpha-expressions to form a long string, with the last character from the first expression preceding the first character from the second expression. Represented in BASIC by the & operator; see Section 3.5.

constant: An item in the BASIC language whose value is self-defining and not subject to change. In this manual, constants are classified by type.

Floating-point constants: See Section 2.2
Integer constants: See Section 2.2
Alpha constants (literals): See Section 3.3

EDITOR: The utility program which generates a source file from keyboard input. The operation of the EDITOR is discussed in Section 1.4, and in the _2200VS Programmer's Introduction_.

element: An element of an array is one of the individual variables which comprise that array. The word "element" is also used to refer to an item in the BASIC language which may appear in a particular situation.

expression: A general term for an item in the BASIC language constructed from operands, operators, and functions according to proper syntax. In this manual, expressions are classified by type.

Floating-point expressions: see Section 2.4
Integer expressions: see Section 2.4
Alpha-expressions: see Section 3.4

FAC: Field Attribute Character. If a character greater than HEX(7F) is displayed on the screen, it will affect the display mode (intensity, blinking, underlining, etc.) and data entry mode of all characters rightward to either the next FAC or the end of the line, whichever comes first. See Section 5.2.

file: A set of data on disk; the unit of data by which a BASIC program gains access to data on disk. See Section 6.2.

floating-point: One of the three BASIC data types; capable of representing a number, with fraction and exponent. See Section 2.1

function: An item in the BASIC language which takes as input one or more expressions and produces a value which can be used in further computations by the program, such as ARCTAN(x), ROUND(x,y), and VAL(x), where x and y are appropriate expressions. The BASIC function repertoire is listed in Section 4.5.

integer: One of the three BASIC data types; capable of representing a whole (nonfractional) number. See Section 2.1.

keyword: A sequence of letters which are interpreted as a unit by the BASIC compiler. Examples of keywords are: LET, IF, THEN, ARCTAN.

leading: Instances of a character occurring to the left of all instances of any other character. There are two leading zeros in "003320".

length: The "defined length" of an alpha value is the number of characters it contains. For variables and array strings, this is defined in the COM, DIM, or MAT REDIM statements. The "actual length" of an alpha variable or array string is the length of the string excluding trailing blanks. An all blank string, however, has an actual length of one. See Section 3.2.

library: A collection of files on disk. Either the BASIC program or the operator at execute-time must specify the name of the library in order to access a file from that library. See Section.

literal: An alpha constant; typically a sequence of characters enclosed in single- or double-quotes, or using the HEX notation. "hello", 'ABCDEF', and HEX(6162CF0F) are literals. See Section 3.3.

matrix: A two-dimensional array; an array for which two numeric expressions are required to uniquely identify an element. Arrays are discussed in Section 2.3.

numeric: Integer or floating-point; as opposed to alpha.

object file: A file containing code which can be executed directly by the 2200 VS processor. The BASIC compiler produces an object file.

operand: A constant or variable, or any expression "operated on"-- taken as input--by a statement, operator, or function.

operator: A symbol which combines two expressions, indicating that, at execute-time, their values should be combined using a particular function. + is an operator signifying the addition function; A+B is an example of that operator with A and B as its operands. See Section 1.11.

receiver: An item which, in a given situation, is capable of receiving and storing a new value. Variables and STR expressions (where the argument to the STR is a receiver) are examples of valid receivers.

scalar: A variable which is not an array or array element.

source file: A file which contains text; for example, a file which contains statements in the BASIC language. The format of a BASIC source file is discussed in Section 1.3.

statement: The smallest unit in the BASIC language capable of invoking a complete action; an item in the BASIC language which may occur in a source file on its own. The correspondence between statements and source-file lines is discussed in Section 1.10; the repertoire of BASIC statements is completely given in Part II.

string: A sequence of characters. String data is contained in a BASIC alpha variable. See Chapter 3.

subscript. A numeric expression used to select one of the elements of an array. Depending on the array, either one or two subscripts may be required uniquely to select an element.

substring: A portion of an alpha variable, defined by the relative starting position and length, using a STR expression. For example, STR(A$,4,3): The substring consisting of three characters from A$ beginning at the 4th. See discussion of STR in Part II.

trailing: Instances of a character in a string to the right of all instances of any other characters. There are two trailing blanks in "ABC ". The zeros in the number "200" would not be called trailing zeros; in numbers, "trailing" also implies occurring to the right of a decimal point.

type: The three BASIC data types are alpha(numeric), floating-point, and integer. A variable's type indicates the type of values it may contain.

value: The value of an expression is the string or number obtained when the listed operands are combined using the specified operators and functions; the value of a variable is the current contents of that variable (the string or number which that variable stands for at the present time), which is the value most recently assigned to it.

variable: A named area of memory whose value may change during the execution of a program. In this manual, variables are classified by type:

    Floating-point variables; see Section 2.1
    Integer variables: see Section 2.1
    Alpha variables: see Section 3.2

variable name: An item in the BASIC language which stands for the current value of a particular variable.

vector: An array which requires only one subscript to uniquely select an element.

volume: A physical disk or tape on the VS system, or the name used by the system to refer to that disk or tape. A BASIC program, or the operator at execute-time, must specify the name of the volume in order to access a file on that volume.

APPENDIX B

2200T AND 2200VP CONSIDERATIONS

ADDITIONAL STATEMENTS

HEXPRINT

```
General Form:

         {alpha variable         } [ {,}{alpha variable      }]
HEXPRINT {alpha array designator} [{;}{alpha array designator}]...[;]
```

Purpose

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). Defined length of the alpha values are printed. Arrays are printed one element after another with no separation characters. A carriage return occurs after the value(s) of each alpha variable (or array) in the argument list, unless the argument is followed by a semi-colon. 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 arguments which are lengthy.

Example:

```
10 A$="ABC"
20 PRINT "HEX VALUE OF A$=";
30 HEXPRINT A$
```

Output: HEX VALUE OF A$=41424320202020202020202020202020

Examples:

```
100 HEXPRINT A$,B$(1), STR(C$,3,4)
110 HEXPRINT A$;B$;
120 HEXPRINT X$()
```

PACK

```
General Form:

PACK (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 negative.
   Bit 2 OFF ("0").
   Bit 3 OFF ("0").
   Bit 4 (rightmost) set to "1" if number 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.

Examples of storage requirements:

```
####       = 2 bytes
###        = 2 bytes
+##.###    = 3 bytes
+#.##↑↑↑↑  = 3 bytes
```

Examples of syntax:

```
000010 PACK (####)A$ FROM X
000040 PACK (####)STR(A$,4,2)FROM N(1)
000070 PACK (##.##) A1$() FROM X,Y,N(),M()
```

UNPACK

```
General Form:

UNPACK (image) alpha-expression TO
                                 [                                ]
{numeric array-designator} |,{numeric array-designator}|...
{numeric variable          } | {numeric variable          }|
                                 [                                ]

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).

Examples of syntax:

```
000010 UNPACK (####)A$ TO X,Y,Z
000020 UNPACK (+#.##)STR(A$,4,2) TO X
000030 UNPACK (+#.##↑↑↑↑)A$() TO N()
000040 UNPACK (######)A$() TO X,Y,N(),M()
```

Example:

```
000010 X=24:DIM A$3
000020 PACK (####)A$ FROM X
000030 PRINT X
000040 PRINT HEXOF (A$)
000050 UNPACK(####)A$ TO Y
000060 PRINT A$,Y
```

Output:

```
24
002420
$                24
```

$PACK/$UNPACK

```
General Forms:

           [                   ]
    $PACK |([{D=}]alpha-exp)]|alpha-receiver FROM arg[,arg]...
           | [{F=}]            |
           [                   ]

                 [,DATA {GOTO } line number]
                        {GOSUB}

where:  line number = line number of 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}

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. Note that an <u>arg</u> of the form "name$()" is <u>always</u> recognized as an array of elements, <u>never</u> as an alpha array string. Use a STR to get <u>that</u> result.

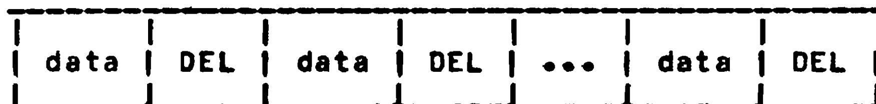
2. Array elements are generally considered as individual consecutive values or receivers (row-by-row). The exception is F format, where a single format applies to all of the array elements.

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 <u>any</u> of the formats.

4. $UNPACK generates an error (or exit) if the unpacked data is not the same type (alpha, numeric) as the receiver.

<u>Delimiter Format</u>

Indicated by the presence of "D = alpha-expression". The format of the $PACK'ed 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 2 = (DEL) delimiter character

byte 1 = conversion code. This is used only by $UNPACK, but must have one of the four legal values for <u>either</u> $PACK <u>or</u> $UNPACK:

Hex Value	(UNPACK) Result
00	A) Error if insufficient data in the buffer. B) Skip a receiver (or array element) for each <u>extra</u> 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 <u>extra</u> 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 - <u>defined</u> 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, trailing, and intervening 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.

> **NOTE:**
>
> $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.

> **NOTE:**
>
> 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.

$PACK/$UNPACK

Field Format

Indicated by the presence of "F = alpha-expression". The format of the $PACK'ed data is:

```
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. (Note: 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. 1D 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. $2h_p$ ASCII Implied decimal

Form:

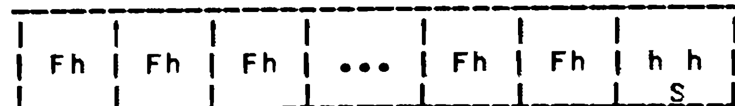
d = (ASCII) digit 0-9
s = sign byte

$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. $3h_p$ IBM numeric display format

Form:



h = hexdigit 0-9 only
h_s = sign digit

$PACK: format as shown

h_s = {C (+) {D (-)

$UNPACK: format as shown; F's are ignored. h_s may be {A,C,E,F (+) {B,D (-)

6. $4h_p$ WANG VS display format

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; 3's ignored

h_s = {D (-)
{all else (+)

7. $5h_p$ IBM packed decimal format

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 more or less 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 $UNPACK'ed, 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_p in the format specification denotes the number of 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:

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:

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

APPENDIX C

VS Field Attribute Characters

BRIGHT	MODIFY	ALL	NOLINE	80
BRIGHT	MODIFY	ALL	LINE	A0
BRIGHT	MODIFY	UPPERCASE	NOLINE	81
BRIGHT	MODIFY	UPPERCASE	LINE	A1
BRIGHT	MODIFY	NUMERIC	NOLINE	82
BRIGHT	MODIFY	NUMERIC	LINE	A2
BRIGHT	PROTECT	ALL	NOLINE	84
BRIGHT	PROTECT	ALL	LINE	A4
BRIGHT	PROTECT	UPPERCASE	NOLINE	85
BRIGHT	PROTECT	UPPERCASE	LINE	A5
BRIGHT	PROTECT	NUMERIC	NOLINE	86
BRIGHT	PROTECT	NUMERIC	LINE	A6
DIM	MODIFY	ALL	NOLINE	88
DIM	MODIFY	ALL	LINE	A8
DIM	MODIFY	UPPERCASE	NOLINE	89
DIM	MODIFY	UPPERCASE	LINE	A9
DIM	MODIFY	NUMERIC	NOLINE	8A
DIM	MODIFY	NUMERIC	LINE	AA
DIM	PROTECT	ALL	NOLINE	8C
DIM	PROTECT	ALL	LINE	AC
DIM	PROTECT	UPPERCASE	NOLINE	8D
DIM	PROTECT	UPPERCASE	LINE	AD
DIM	PROTECT	NUMERIC	NOLINE	8E
DIM	PROTECT	NUMERIC	LINE	AE
BLINK	MODIFY	ALL	NOLINE	90
BLINK	MODIFY	ALL	LINE	B0
BLINK	MODIFY	UPPERCASE	NOLINE	91
BLINK	MODIFY	UPPERCASE	LINE	B1
BLINK	MODIFY	NUMERIC	NOLINE	92
BLINK	MODIFY	NUMERIC	LINE	B2
BLINK	PROTECT	ALL	NOLINE	94
BLINK	PROTECT	ALL	LINE	B4
BLINK	PROTECT	UPPERCASE	NOLINE	95
BLINK	PROTECT	UPPERCASE	LINE	B5
BLINK	PROTECT	NUMERIC	NOLINE	96
BLINK	PROTECT	NUMERIC	LINE	B6
BLANK	MODIFY	ALL	NOLINE	98
BLANK	MODIFY	ALL	LINE	B8
BLANK	MODIFY	UPPERCASE	NOLINE	99
BLANK	MODIFY	UPPERCASE	LINE	B9
BLANK	MODIFY	NUMERIC	NOLINE	9A
BLANK	MODIFY	NUMERIC	LINE	8A
BLANK	PROTECT	ALL	NOLINE	9C
BLANK	PROTECT	ALL	LINE	BC
BLANK	PROTECT	UPPERCASE	NOLINE	9D
BLANK	PROTECT	UPPERCASE	LINE	BD
BLANK	PROTECT	NUMERIC	NOLINE	9E
BLANK	PROTECT	NUMERIC	LINE	BF

APPENDIX D

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. If PMAP = STATIC, a complete PMAP is produced, 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 (if PMAP = STATIC), beginning with *COMMON on a new page. In the common area map the columns serve the same purposes 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 (PMAP = STATIC) 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 purposes as in the PMAP, with the exception of the first column, which contains either *STATIC, indicating the address of the contents of the Static section follow, or *PGT (Program Global Table) indicating the addresses 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 DEFFN' 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. (In this case, 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.

ERRLIST

If ERRLIST = YES, a listing of the compiler diagnostics is produced.

SUBCHK

If SUBCHK = YES, the compiler generates special code which checks the ranges of subscripts during program execution, and causes a program check (execution interruption) if a subscript exceeds its defined limit. Otherwise, no check is performed on subscripts during execution.

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 greater than or equal to the specified FLAG value will cause the compiler to print a diagnotic 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 sets the number of lines per page for all compiler produced printouts.

APPENDIX E **VS CHARACTER SET**

NOTE: *b_7 always equals zero*.*				b_6 →	0	0	0	0	1	1	1	1
				b_5 →	0	0	1	1	0	0	1	1
				b_4 →	0	1	0	1	0	1	0	1
b_3 ↓	b_2 ↓	b_1 ↓	b_0 ↓	High-Order Digit → / Low-Order Digit ↓	0	1	2	3	4	5	6	7
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 F <u>ASCII COLLATING SEQUENCE</u>

This appendix lists all displayable characters in the Wang VS character set, with hexadecimal equivalents. The list is arranged in the order of the machine's collating sequence.

Order in ASCII Collating Sequence	ASCII Character	Hexadecimal Equivalent	Order in ASCII Collating Sequence	ASCII Character	Hexadecimal Equivalent
01	■(pseudoblank)	0B	47	M	4D
02	space	20	48	N	4E
03	!	21	49	O	4F
04	"	22	50	P	50
05	#	23	51	Q	51
06	$	24	52	R	52
07	%	25	53	S	53
08	&	26	54	T	54
09	'(quote)	27	55	U	55
10	(	28	56	V	56
11	)	29	57	W	57
12	*	2A	58	X	58
13	+	2B	59	Y	59
14	,	2C	60	Z	5A
15	-(minus)	2D	61	[	5B
16	.	2E	62	\	5C
17	/	2F	63	]	5D
18	0	30	64	↑	5E
19	1	31	65	←	5F
20	2	32	66	°(degree)	60
21	3	33	67	a	61
22	4	34	68	b	62
23	5	35	69	c	63
24	6	36	70	d	64
25	7	37	71	e	65
26	8	38	72	f	66
27	9	39	73	g	67
28	:	3A	74	h	68
29	;	3B	75	i	69
30	<	3C	76	j	6A
31	=	3D	77	k	6B
32	>	3E	78	l	6C
33	?	3F	79	m	6D
34	@	40	80	n	6E
35	A	41	81	o	6F
36	B	42	82	p	70
37	C	43	83	q	71
38	D	44	84	r	72
39	E	45	85	s	73
40	F	46	86	t	74
41	G	47	87	u	75
42	H	48	88	v	76
43	I	49	89	w	77
44	J	4A	90	x	78
45	K	4B	91	y	79
46	L	4C	92	z	7A

APPENDIX G BASIC KEYWORDS

The following is a partial glossary of BASIC keywords, including all verbs, functions, and operators.

ABS function: returns the absolute value of an expression.

ACCEPT: allows formatted input of data from the workstation, with field verification and/or range checking.

ADD: adds binary values of two arguments in a logical expression, one byte at a time, and places result in alpha-receiver.

ADDC: like ADD, but treats the two arguments as multi-byte binary numbers.

ALL function: used in logical expressions to generate an argument consisting entirely of the same specified character (similar to INIT).

AND:

1. logically AND's two arguments, one byte at a time.
2. AND of two boolean subexpressions in an IF statement.

ARCCOS function: returns the arccosine of an expression.

ARCSIN function: returns the arcsine of an expression.

ARCTAN function: returns the arctangent of an expression.

ATN function: same as ARCTAN.

BIN function: converts integer value of an expression to an alphanumeric value which is the binary equivalent of the expression. Inverse of BIN function.

BOOLh: does one of 16 possible logical (boolean) operations, depending upon the value of h.

CALL: calls external subroutine marked by SUB, which must have been linked to main program.

CLOSE: closes file previously opened by OPEN.

COM: establishes common storage area for variables used by more than one program. Like DIM, it reserves space for arrays and sets length for alpha variables.

Concatenation operation (&): combines two strings, the second being put directly after the first without intervening characters. The result is treated as a single string.

CONVERT:

1. converts number represented by ASCII characters in alphanumeric expression to a numeric value, and sets a numeric variable equal to that value.
2. converts numeric value to an ASCII character string representing it, and places string in an alpha-receiver in a specified format.

COPY: transfers an alpha-expression to an alpha receiver, one byte at a time.

COS function: returns the cosine of an expression.

DATA: provides data values which can be used by variables in a READ statement, enabling constants to be stored within program.

DATE function: returns a six-character alpha string giving the current date.

DEFFN: defines a single-valued user-written numeric function, referenced by FN.

DEFFN':

1. defines PF key or program entry point for subroutine with argument-passing capability.
2. defines literal to be supplied for text entry when PF key is used.

DELETE: in disk I/O, deletes last record READ. Valid only for indexed files.

DIM: reserves space for arrays and sets length for alpha variables.

DIM function: returns, as integer value, the current row or column dimension of an array.

DISPLAY: allows formatted output of data values on screen.

END: terminates program prior to physical end; can pass program-supplied return code to system for use in Procedures.

EXP function: finds the value of e raised to the value of the expression.

FMT: specifies data formats for PRINTUSING and File I/O statements.

FN function: calls a function previously defined in a DEFFN statement.

FOR: initiates a loop ending with a NEXT statement.

FS system function: returns file status for the previous I/O operation on the specified file.

GET: extracts data from an I/O buffer or from an alpha-expression.

GOSUB: transfers control to first program line of an internal subroutine.

GOSUB': transfers control to an internal subroutine (marked by DEFFN'); unlike GOSUB, can pass arguments.

GOTO: transfers control to specified line number.

HEX literal string: allows the user to supply ASCII code for characters for which no keyboard characters exist, including field attribute characters.

HEXPACK: converts an ASCII character string representing a string of HEX digits into the binary equivalent of those digits.

HEXPRINT: prints the value of an alpha variable or the values of an alpha array in hexadecimal notation. The same effect as PRINT HEXOF. Supported in VS BASIC only for compatibility with the 2200T.

HEXUNPACK:	converts the binary value of an alpha-expression to a string of ASCII characters representing the hexadecimal equivalent of that value.
IF...THEN... ELSE	tests relation and causes conditional transfer or statement execution based on result of test.
Image(%):	used with PRINTUSING or Disk I/O to format output.
INIT:	sets all characters in one or more alpha-receivers equal to first character of an alpha-expression.
INPUT:	1. allows user to supply data during program execution. 2. in conjunction with DEFFN' statement, allows user to enter defined text or branch to marked subroutine by means of PF keys.
INT function:	returns the largest integer less than or equal to the value of an expression.
KEY system function:	returns the primary (or alternate) key of the last record read in disk I/O.
LEN function:	determines actual length, in bytes, of alpha-expression.
LET:	assigns the value of an expression to one or more receivers.
LGT function:	returns the logarithm base ten of an expression.
LOG function:	returns the natural logarithm of an expression.
MASK system function:	returns the alternate-key mask for the last record read in disk I/O.
MAT +:	adds two arrays of the same dimension.
MAT ASORT:	sorts array in ascending order.
MAT CON:	sets all elements of array to 1. Can also redimension array.
MAT DSORT:	sorts array in descending order.

MAT =: replaces each element of one array with corresponding element of another array. Redimensions first array to conform to second array.

MAT IDN (MAT identity): causes specified matrix to assume form of matrix identity.

MAT INPUT: allows user to supply values for an array from the workstation during program execution.

MAT INV (MAT inverse): causes one matrix to be replaced by inverse of another matrix.

MAT *: stores product of two arrays in a third array.

MAT PRINT: prints arrays.

MAT READ: assigns values contained in DATA statements to array variables without referencing each member of the array individually.

[MAT] REDIM: redimensions an array.

MAT ()* (MAT scalar multiplication): multiplies each element in an array by an expression. Result is stored in second array or in same array.

MAT - (MAT subtraction): subtracts numeric arrays of same dimension.

MAT TRN (MAT transpose): causes one array to be replaced by transpose of another array.

MAT ZER: sets all elements of array to zero. Can redimension array.

MAX function: returns maximum value in numeric list.

MIN function: returns minimum value in numeric list.

MOD function: returns value of first expression modulo second expression.

NEXT: marks end of loop initiated by FOR.

NOT: inverts value of boolean subexpression in an IF statement.

NUM function: counts number of sequential ASCII characters in an alpha-expression that represent a legal BASIC number.

ON...GOTO or GOSUB: computed GOTO or GOSUB statement. Branches to one of a number of lines depending upon the value of expression following ON clause.

OPEN: opens disk file for I/O operation. Must be preceded by SELECT.

OR:
1. logically OR's two arguments in a logical expression.
2. OR of two boolean subexpressions in an IF statement.

PACK: packs numeric values into an alphanumeric receiver, reducing storage requirements. Included in VS BASIC for compatibility with the 2200T.

#PI function: assigns the value 3.14159265358979323.

POS function: searches an alpha-expression for a character that fits a defined relationship to another alpha expression, and outputs this character's position.

PRINT:
1. sends output to printer or workstation (as chosen by SELECT).
2. controls printer position, workstation display location, and workstation bell.

PRINTUSING: sends formatted output to workstation or printer; format determined by referenced IMAGE or FMT statement.

PUT: inserts data into I/O data buffer or alpha-receiver.

READ: in conjunction with DATA, assigns elements in DATA list to receivers in READ list.

READ disk file: causes one record from disk file to be read, either into record area, or, with an argument list, into both record area and arguments.

REM: denotes comment; remainder of statement is ignored by system.

RESTORE: allows repetitive use of DATA statement values by READ statements, by setting DATA pointer back to specified DATA value.

RETURN: used in a subroutine to return processing of program to the statement following the last executed GOSUB or GOSUB' statement.

RETURN CLEAR: used in a subroutine to clear subroutine return address information from memory. Execution continues with statement following RETURN CLEAR.

REWRITE: used to rewrite an existing record, which must have already been read with the HOLD option.

RND function: produces a pseudorandom number between 0 and 1.

ROTATE [C]: rotates bits in an alpha-receiver.

ROUND function: rounds an expression to a specified number of decimal places.

SEARCH: searches alpha-expression for strings of the same length as a second alpha-expression which satisfy one of the following relations:

{> }
{>=}
{< }
{<=}
{= }
{<>};

and places starting positions of substrings satisfying the relationship into numeric array or alpha receiver.

SELECT:

1. routes output to printer or workstation.
2. specifies whether arguments and results of trig functions are to be in degrees, radians, or gradians.

3. specifies execution pause after each write to workstation.

4. specifies the size of a program's buffer pool and allows the pool to be shared by several files.

SELECT disk file: sets up file information and record area for a file, which can later be opened for input or output by an OPEN statement.

SGN function: returns the value 1 if the argument is any positive number, 0 if the argument is zero, and -1 if the argument is any negative number.

SIN function: returns the sine of an expression.

SIZE system function: returns, as an integer, the size of the last record read from a specified file.

SKIP: positions a consecutive file forward or backward a given number of positions.

SQR function: finds the square root of an expression.

STOP: interrupts program execution, until 1) ENTER is keyed, or 2) a PF key corresponding to a subroutine marked by a DEFFN' causes program to continue at the entry point of subroutine.

STR function: specifies a substring of an alpha variable or alpha array string. With it, a portion of an alpha value can be examined, extracted, or changed.

SUB: defines an external subroutine, called from a separate BASIC (or other) program, and which must be linked to the calling program before being run.

TAN function: returns the tangent of an expression.

TIME function: returns an eight-character alpha string giving time of day.

TRACE: traces program execution, producing output on printer.

TRAN: translates (in place) the characters in an alpha-receiver, via an alpha-expression which is used as a translate table or list.

UNPACK: unpacks data that was packed by the PACK statement. UNPACK is supported in VS BASIC for compatibility with the 2200T.

VAL function: converts alphanumeric expression to integer value which is the binary equivalent of the expression. Inverse of BIN function.

WRITE: writes the next sequential record to a file, using data in record area, or, if argument list is present, argument list.

XOR:

1. logically exclusive OR's two arguments in a logical expression.
2. exclusive OR of two boolean subexpressions in an IF statement.

INDEX

Page

ABS (absolute value) function . . . . 29 ,180
ACCEPT statement 78
ADD[C] 46
ADDR-type subroutines 63, 114, 223
addition 15
   array 95, 162
   priority of 15
addition array statement 95, 162
ALL 50, 108
alphanumeric
   array 37
      defining 38
      length of elements 38
      naming an 37
      array strings 43
   constants . . . . (see literal strings)
   data
      in FMT statement 73, 135
      in Image statement 74, 151
      in INPUT statement 154
      in PRINT statement 192
      in scalar assignment statement (LET) 159
      in array assignment statement 94, 165
      logical operator with 44, 109, 111, 190, 231
      relational operators with 149
   expressions 41
   functions 47
   literal strings 40
   operators (see also alphanumeric data,
   logical operators with) 43, 44
   variables 37
AND logical operator 45, 109
ARCCOS (arc cosine) function 28, 180
ARCSIN (arc sine) function 28, 180
ARCTAN (arc tangent) function 28, 180
argument
   in user-defined functions 123, 137
arithmetic . . . . (see numeric)

Page

array assignment statement 94, 165
operations with
addition 94, 177
identity function 94, 166
inverse function 94, 169
matrix multiplication 94, 171
sort
ascending sort 94, 163
descending sort 94, 163
subtraction 94, 177
transpose function 94, 178
redimensioning arrays with 94
arrays 22, 37
alphanumeric 37
comparison between one- and two-dimensional 23
default dimensions 23
defining 25
COM Statememt 117
DIM statement 130
elements of 21, 37
expressions in 26, 41
initial value of 21, 37
input values for, through INPUT statement 167
naming 21, 41
numeric 21
output values from
with ACCEPT statement 78, 100
with DISPLAY statement 77, 132
with MATPRINT statement 172
with PRINT statement 192
redimensioning 94
subscripts 21
ASORT statement 163
assignment statement
array (see MAT=) 165
scalar (see LET) 159
ATN (arc tangent) function 28, 180

B, insertion character in FMT 134
BASIC character set 254
BASIC statements 12
BIN function 110

Page

binary operators . . . .(see operators)
blank lines, printing 73, 194
blanks
   as digit specifier in FMT statement 134
   ignored by BASIC 14
   in Image statement 151
   in literal strings 39
   initial value of alphanumeric array 38
   null delimiter . . . . . . (see comma, used as null delimeter)
BOOLh logical operator 45, 111
branching
   program 57, 143, 149, 185
   subroutine 58, 133, 141, 142, 203, 221
built-in functions (see functions) 27, 47

CALL statement 113
CH(w) data specification 134
character . . . . (see alphanumeric)
character set, BASIC 254
   ASCII collating sequence of 255
CLOSE statement 82, 116
closing files (see CLOSE statement) 116
colon (:), to separate statements 14
COM statement 117
comma (,)
   as a data separator 121, 154
   as an insertion character in FMT statement 134
   to specify full print zones 192
   use in INPUT statement 154
   use in PRINT statement 192
   used as null delimiter during input 154
comment
   in * statement 13
   in REM statement 201
computed GOSUB statement 185
computed GOTO statement 185
concatenation (&) 43
consecutive files 85
constants
   alphanumeric literal strings 40
   internal . . . . (see #PI and e)
   numeric 20
continuation character, use of exclamation point as 17
control specification, in FMT statement 134
   (see also POS, SKIP, and X)
control variable, in FOR statement (see index variable) . 138, 183

Page

CONVERT statement 118
COPY statement 120
COS (cosine) function 28, 180

data . . . . (see alphanumeric and numeric)
data form specification, in FMT and ACCEPT statements
  (see also CH and PIC) 73, 78, 100, 134
DATA statement 121
  relationship to READ statement 198
  relationship to RESTORE statement 202
DATE function 122
decimal point (.)
  in Image statement 151
  in PRINT statement 192
  insertion character in FMT statement 134
default values
  of alphanumeric variables 38
  of numeric variables 21
DEFFN statement 123
DEFFN' statement 60, 125
DELETE statement 129
  restriction with consecutive files 129, 212
delimiters . . . . (see PRINT,INPUT)
descending sort function array assignment statement . (see DSORT)
digit specifiers
  in FMT statement 134
  in Image statement 151
DIM function 131
DIM statement 130
  defining alphanumeric data with 38, 117, 130
  defining arrays with 25, 38
dimensions, of arrays 25, 38, 117, 130
DISPLAY statement 77, 132
division 15
  priority of 15
dollar sign ($)
  floating character in FMT 134
  to name alphanumeric variables 37
DSORT statement 163

Page

dummy variable 123, 137
relationship to argument 137

E as an exponential specifier 20
ELSE keyword in IF Statement 149
end of file exit
in GET statement 90, 140
in READ file statement 90, 198
in SELECT statement 90, 212
END statement 133
EOD keyword
in GET statement 90, 140
in PUT statement 90, 197
in READ file statement 90, 198
in REWRITE file statement 90, 205
in WRITE file statement 90, 229
error conditions
in GET statements 90, 140
in PUT statements 90, 197
in READ file statement 90, 198
in REWRITE file statement 90, 205
in WRITE file statement 90, 229
evaluation
of logical expressions and subexpressions 45
of numeric expressions 15, 26
executable statements . . . . (see nonexecutable statements)
EXP (natural exponential) function 29, 180
explicit declaration
of array dimensions 37
of alpha variable length 38
exponent specifier
in FMT statement 134
in Image statement 151
exponentiation
operator 15
priority of 15
expressions
alphanumeric 42
logical 44
numeric 26
evaluation of 15, 26, 45
testing in IF statement 149

Page

false value, in IF statement 149
file input/output statements 82
CLOSE file 116
DELETE file 129
OPEN file 186
READ file 198
REWRITE file 205
WRITE file 229
filename, definition of 84
files 82
naming conventions for 84
final value, in FOR statement 138
FL[(w)] format specification 134
floating characters in PIC 100
floating-point constant (E-format) 20
FMT statement 73, 134
used with PRINTUSING 73, 196
used with files 197, 205, 229
FN, to identify user-defined functions 137
FOR statement 138
format control specifications in FMT statement
(see also X, POS, SKIP) 134
FS function 89, 139
functions 27, 47
user-defined 137
DEFFN statement to define 123

GET statement 140
error handling with 92, 140
GOSUB statement 59, 141
computed GOSUB . . . . (see ON statement)
GOSUB' statement 59, 142
GOTO keyword in IF statement 149
GOTO statement 143
computed GOTO (see ON statement) 185
used with IF statement 149

HEX digit 44
HEX literal string 144
hexadecimal 44
HEXPACK statement 145
HEXUNPACK statement 148
HOLD keyword in OPEN file statement 186

identity function, array statement 166
IDN identity array statement 166
IF statement 149
logical operators in 149

Page

Image (%) statement 73, 151
  general format of 151
implicit declaration . . . . (see default values)
increment value, in FOR statement 138, 183
index variable in FOR statement 138, 183
indexed files 85, 89
  keys in 85, 89
  reading records from 87, 199
  writing records into 87, 205, 229
  rules for 89
INIT statement 153
initial value, in FOR statement 138
INPUT statement 70, 154
  compared to workstation I/O statements 72
  delimiters 154
  use with arrays . . . . (see MAT INPUT)
input/output statements
  ACCEPT 78, 100
  DISPLAY 132
  INPUT 154
  PRINT 192
  file-oriented (see also entries for the individual
    statements listed below)
    CLOSE file 116
    DELETE file 129
    OPEN file 186
    READ file 198
    REWRITE file 205
    WRITE file 229
insertion character
  in FMT specification 134
  in PIC specification 100
INT (integer) function 30, 180
INV inverse array statement (see MAT INV) 169
IOERR keyword 91
  in DELETE file statement 129
  in GET statement 140
  in PUT statement 197
  in READ file statement 198
  in REWRITE file statement 205
  in WRITE file statement 229

Page

key, indexed . . . .(see indexed files)
KEY clause
in DELETE file statement 129
in READ file statement 198
in REWRITE file statement 205
key-sequenced files . . . .(see indexed files)
KEY (key position) function 89, 157

LEN (length of character string) function 158
LET statement 159
LGT (logarithm to base 10) function 30
line numbers 17
line skipping, in printed lines 192
LOG (logarithm to base e) function 30, 180
logical expressions 44
logical operators 46, 109, 111, 190, 231
loops 57
FOR and NEXT statements 138, 183

MASK Function 161
MAT + ... 95, 162
MAT ASSORT/DSORT 95, 163
MAT CON 94, 164
MAT = ... 94, 165
MAT IDN 94, 166
MAT INPUT 94, 167
MAT INV 95, 169
MAT * 95, 171
MAT PRINT 94, 172
MAT READ 94, 173
MAT REDIM 95, 175
MAT ()* 95, 176
MAT - 95, 177
MAT TRN 94, 178
MAT ZER 95, 179
matrix inverse (see MAT INV) 95, 169
matrix multiplication (see MAT*) 95, 171
MAX (maximum value) function 30, 180

Page

member, array . . . . (see array, element of)
MIN (minimum value) function 30, 180
minus sign (-)
   as a binary operator 15
   in FMT 134
   in PIC 100
multiplication 16
   array (see MAT*) 171
   operator 15
   priority of 15

names
   array 22, 37
   file 83
   variable 21, 37
negative increments, in FOR statement 138
nested function references 123
nested loops 138
NEXT statement 183
nonexecutable statements 13
null literal string 40
null delimiter
   consecutive commas during input 154
NUM (number of numeric characters in a character string)
   function 184
number sign 134, 151
numeric 19
   array 22
   constants 20
   expressions 26
      evaluation of 27
      relational operators in 16
   functions 27
   operators 15
   signs, in FMT 134
   variables 21
numeric conversion 118, 158
numeric data 19

Page

ON statement 185
one-dimensional array 23,37
compared to two-dimensional array 24
restrictions in redimensioning 94
OPEN statement 86, 186
opening files 86, 186
operators
alphanumeric . . . .(see logical)
logical (also see individual operators) 46
numeric 15
in PRINT statement 192
priority of 15
relational 16
OR logical operator 190
output
formatting printed 69, 134, 151, 196
DISPLAY statement 77, 132
FMT statement 73, 134
Image (%) statement 73, 151
PRINT statement 73, 192
PRINTUSING statement 73, 196
output files . . . . (see print files)
output list
syntax definition of 192

parentheses
in arithmetic operations 15
to define arrays 22, 38
to enclose array expressions 24, 37
to enclose PIC specifications 100
Percent sign (%)
to identify Image statement 151
to identify integers 21
PIC 100
plus sign (+)
as an operator 15
in FMT 134
in PIC 100
POS function 191
pound sign (#)
in FMT statement 134
in Image statement 151
print files 85

Page

PRINT statement 69, 192
  compared to workstation and printer I/O statements 72
  delimiters 192
PRINTUSING statement 196
print zone 73
printing a line 73, 192
priority of arithmetic operators 15
program 4
  entering, running, and saving 4, 11
program termination
  END statement 133
  STOP statement 218
prompting input 69, 78, 100, 154
PUT statement 197

question mark (?)
  to prompt input 69, 154
  in INPUT statement 69, 154
quotation marks (", ')
  to delimit alphanumeric data 40
  to indicate lowercase 40

READ statement 198
  relationship to DATA statement 121, 198
  relationship to RESTORE statement 202
  using with arrays (see MAT READ) 173
READ file statement 198
record, definition of 83
redimensioning arrays 94
relational operators 16
REM statement 201
remarks in program 201
RESTORE statement 202
RETURN statement 56, 203
  used with GOSUB 141
  RETURN CLEAR 204
REWRITE statement 205
  restriction on use 186, 205
RND (random number) function 30, 207
ROTATE statement 208
ROUND statement 30, 209

scalar assignment statement (see LET statement) 159
scalar multiplication . . . . (see multiplication)
SEARCH statement 210
SELECT statement 212
SELECT file statement 213

Page

semicolon (;)
as null delimiter 193
in PRINT statement 193
used to suppress line feed 74, 193
sequential access
of records in key-sequenced files 200
sequential files . . . . (see consecutive files)
SGN (Signum) function 31
SIN (sine) function 28
SIZE function 90, 216
SKIP printer control specification 134, 194
SKIP file control statement 217
slash (/)
in FMT as an insertion character 135
sort statements . . . . (see ASORT, DSORT)
spacing of printed values 192
SQR (square root) function 29
square array needed for identity function 166
STEP keyword
in FOR statement 138
STOP statement 218
STR (portion of string) function 41, 51, 219
SUB statement 62, 221
subroutines
definition of 58
external 58, 61
internal 58, 59
(also see CALL, GOSUB, GOSUB')
subscript 25
substring 219
subtraction 15
syntax, rules of 16

TAN (tangent) function 28
test value, in FOR statement 138
THEN keyword in IF statement 149
TIME function 224
TRACE statement 225
TRAN statement 227
trailing signs, in FMT statement 136
transfer of control
general discussion 56-58
(see also ACCEPT, CALL, FOR, IF, END, INPUT, GOSUB, GOSUB')
transpose function . . . . (see MAT TRN)
truncation of literal string 40
two-dimensional arrays . . . . (see arrays)
restriction in redimensioning 96, 97

Page

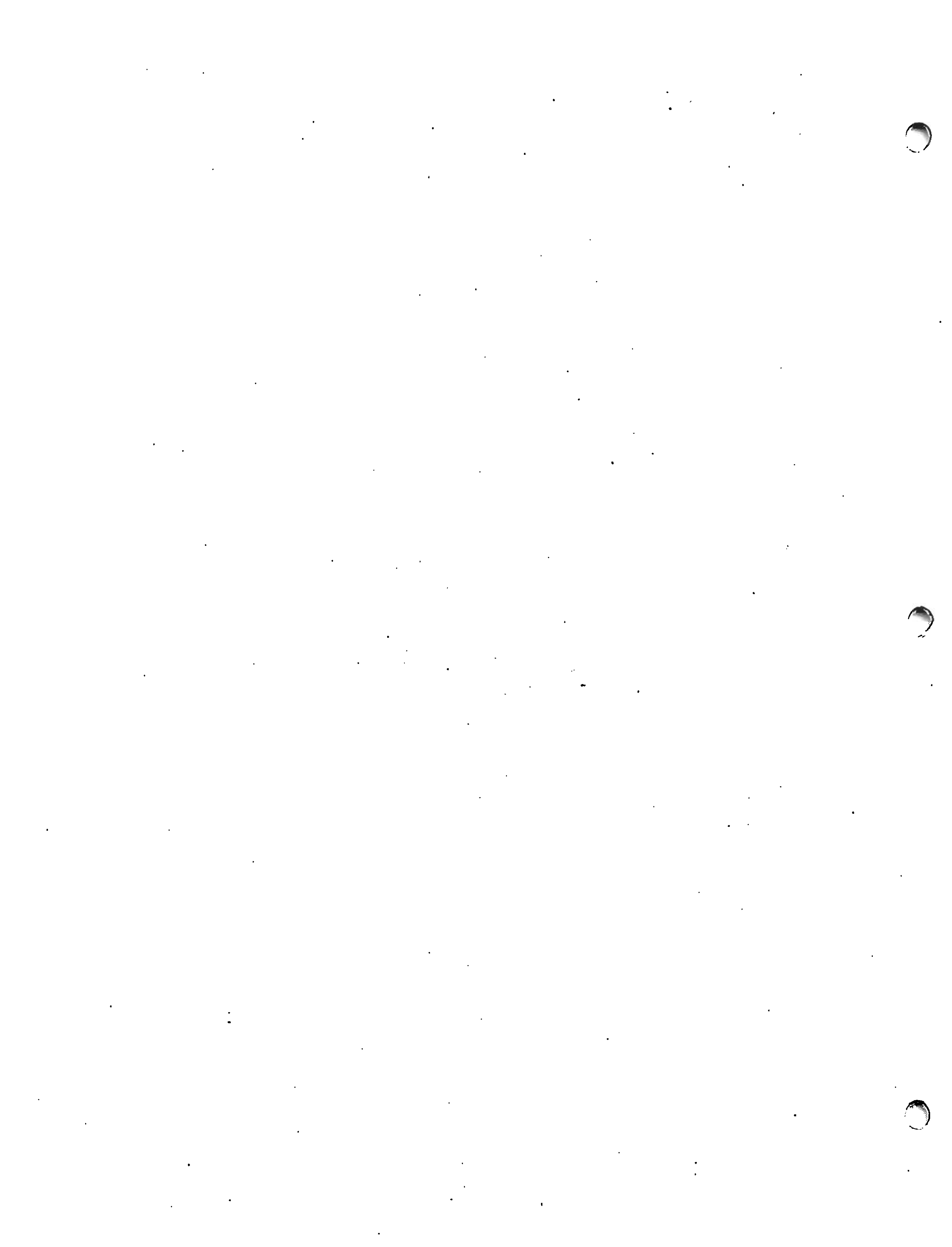
user-written functions (FN) 137
   DEFFN statement to define 123
   maximum number permitted 124
USING clause to relate to FMT or
 Image statement . . . . (see PUT, REWRITE, WRITE)

VAL function 31, 49, 228
variable
   alphanumeric 37
   dummy 123, 137
   naming conventions for 22, 37
   numeric 22
   VS Character Set 254

WRITE file statement. 88, 229

XOR logical operator 46, 231

Zero, as initial value of numeric variable 21
zero suppression
   in FMT statement 135
zones, print 192



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-01

TITLE OF MANUAL VA BASIC REFERENCE MANUAL

COMMENTS:

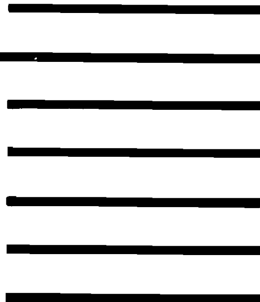
Fold

Fold

(Please tape, Postal regulations prohibit the use of staples.)

WANG

Fold



FIRST CLASS
PERMIT NO. 16
Tewksbury, Mass.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

— POSTAGE WILL BE PAID BY —

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

North America:

Alabama
Birmingham
Mobile

Alaska
Anchorage

Arizona
Phoenix
Tucson

California
Fresno
Inglewood
Los Angeles
Sacramento
San Diego
San Francisco
San Mateo
Sunnyvale
Tustin
Ventura

Colorado
Denver

Connecticut
New Haven
Stamford
Wethersfield

District of Columbia
Washington

Florida
Jacksonville
Miami
Orlando
Tampa

Georgia
Atlanta

Hawaii
Honolulu

Illinois
Chicago
Morton
Park Ridge
Rock Island

Indiana
Indianapolis
South Bend

Kansas
Overland Park
Wichita

Kentucky
Louisville

Louisiana
Baton Rouge
Metairie

Maryland
Rockville
Towson

Massachusetts
Boston
Burlington
Littleton
Lowell
Tewksbury
Worcester

Michigan
Grand Rapids
Okemos
Southfield

Minnesota
Eden Prairie

Missouri
Creve Coeur

Nebraska
Omaha

Nevada
Reno

New Hampshire
East Derry
Manchester

New Jersey
Howell
Mountainside

New Mexico
Albuquerque

New York
Albany
Buffalo
Lake Success
New York City
Rochester
Syracuse

North Carolina
Charlotte
Greensboro
Raleigh

Ohio
Cincinnati
Columbus
Middleburg Heights
Toledo

Oklahoma
Oklahoma City
Tulsa

Oregon
Beaverton
Eugene

Pennsylvania
Allentown
Camp Hill
Erie
Philadelphia
Pittsburgh
Wayne

Rhode Island
Cranston

South Carolina
Charleston
Columbia

Tennessee
Chattanooga
Knoxville
Memphis
Nashville

Texas
Austin
Dallas
Houston
San Antonio

Utah
Salt Lake City

Virginia
Newport News
Richmond

Washington
Seattle
Spokane

Wisconsin
Brookfield
Madison
Milwaukee

Canada
Wang Laboratories (Canada) Ltd.
Don Mills, Ontario
Calgary, Alberta
Edmonton, Alberta
Winnipeg, Manitoba
Ottawa, Ontario
Montreal, Quebec
Burnaby, B.C.

International Subsidiaries:

Australia
Wang Computer Pty. Ltd.
Sydney, NSW
Melbourne, Vic.
Canberra, A.C.T.
Brisbane, Qld.
Adelaide, S.A.
Perth, W.A.
Darwin, N.T.

Austria
Wang Gesellschaft M.B.H.
Vienna

Belgium
Wang Europe, S.A.
Brussels
Erpe-Mere

Brazil
Wang do Brasil Computadores Ltda.
Rio de Janeiro
Sao Paulo

China
Wang Industrial Co., Ltd.
Taipei, Taiwan

France
Wang France S.A.R.L.
Bagnolet
Ecully
Nantes
Toulouse

Great Britain
Wang Electronics Ltd.
Northwood Hills, Middlesex
Northwood, Middlesex
Harrogate, Yorkshire
Glasgow, Scotland
Uxbridge, Middlesex

Hong Kong
Wang Pacific Ltd.
Hong Kong

Japan
Wang Computer Ltd.
Tokyo

Netherlands
Wang Nederland B.V.
Ijsselstein

New Zealand
Wang Computer Ltd.
Grey Lynn, Auckland

Panama
Wang de Panama (CPEC) S.A.
Panama

Republic of Singapore
Wang Computer Pte., Ltd.
Singapore

Republic of South Africa
Wang Computers (South Africa) (Pty.) Ltd.
Bordeaux, Transvaal
Durban
Capetown

Sweden
Wang Skandinaviska AB
Solna
Gothenburg
Arloev
Vasteras

Switzerland
Wang S.A./A.G.
Zurich
Bern
Pully

West Germany
Wang Laboratories GmbH
Berlin
Cologne
Duesseldorf
Fellbach
Frankfurt/M.
Freiburg/Brsg.
Hamburg
Hannover
Kassel
Munich
Nuernberg
Stuttgart

International Representatives:

Argentina
Bolivia
Canary Islands
Chile
Colombia
Costa Rica
Cyprus
Denmark
Dominican Republic
Ecuador
Finland
Ghana
Greece
Guatemala
Iceland
India
Indonesia
Iran
Ireland
Israel
Italy
Jamaica
Japan
Jordan
Kenya
Korea
Lebanon
Liberia
Malaysia
Mexico
Morocco
Nicaragua
Nigeria
Norway
Pakistan
Peru
Philippines
Portugal
Saudi Arabia
Spain
Sri Lanka
Syria
Thailand
Tunisia
Turkey
United Arab Emirates
Venezuela
Yugoslavia

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 851-4111, TWX 710 343-6769, TELEX 94-7421

Printed in U.S.A.
800-1202BA-01
1-79-1.5M
Price: see current list