ATTN: CHARLIE GIBBS

00918
CAV208M45541 UP 9169 R2

UAS

SPERRY UNIVAC
SUITE 906
1177 WEST HASTINGS ST
VANCOUVER BC V6E 2K3

CAV

SPERRY UNIVAC COMPUTER SYSTEMS

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

BASIC

Programmer Reference

UP-9168 Rev. 1–B

This Library Memo announces the release and availability of Updating Package B to "SPERRY UNIVAC Operating System/3 (OS/3) BASIC Programmer Reference", UP-9168 Rev. 1.

This update describes the following BASIC features for release 8.0:

- BATCH END-OF-DATA REACHED message
- BASIC TASK NORMAL TERMINATED message
- RCSZ parameter

All other changes are corrections or expanded descriptions applicable to features present in BASIC prior to the 8.0 release.

Copies of Updating Package B are now available for requisitioning. Either the updating package only or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive only the updating package, order UP-9168 Rev. 1–B. To receive the complete manual, order UP-9168 Rev. 1.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ and MZ	Mailing Lists A00, A16, B00, B16, 18, 18U, 19, 19U, 20, 20U, 21, 21U, 28U, 29U, 75, 75U, 76, and 76U (Package B to UP-9168 Rev. 1, 24 pages plus Memo)	Library Memo for UP-9168 Rev. 1–B
		RELEASE DATE: September, 1982

SPERRY UNIVAC
COMPUTER SYSTEMS

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

BASIC

Programmer Reference

UP-9168 Rev. 1-A

Horst

This Library Memo announces the release and availability of Updating Package A to "SPERRY UNIVAC Operating System/3 (OS/3) BASIC PROGRAMMER Reference", UP-9168 Rev. 1.

This update for release 7.1 includes additional information about workstation screens for BASIC.

Copies of Updating Package A are now available for requisitioning. Either the updating package only or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive only the updating package, order UP-9168 Rev. 1-A. To receive the complete manual, order UP-9168 Rev. 1.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ (less DE, GZ, HA) MZ, 18U, 19U, 20U, 21U, 28U, 29U, 75U and 76U	Mailing Lists DE, GZ, HA, 18, 19, 20, 21, 75 and 76 (Package A to UP-9168, Rev. 1, 11 pages plus Memo)	Library memo for UP-9168 Rev. 1-A
		RELEASE DATE: December, 1981

SPERRY UNIVAC
COMPUTER SYSTEMS

PUBLICATIONS REVISION

Operating System/3 (OS/3)

BASIC

Programmer Reference

UP-9168 Rev. 1

This Library Memo announces the release and availability of "SPERRY UNIVAC® Operating System/3 (OS/3) BASIC Programmer Reference", UP-9168 Rev. 1.

This revision includes minor corrections and expanded descriptions applicable to features present in BASIC prior to the 7.1 release.

Destruction Notice: If you are going to OS/3 release 7.1, use this revision and destroy all previous copies. If you are not going to OS/3 release 7.1, retain the copy you are now using and store this revision for future use.

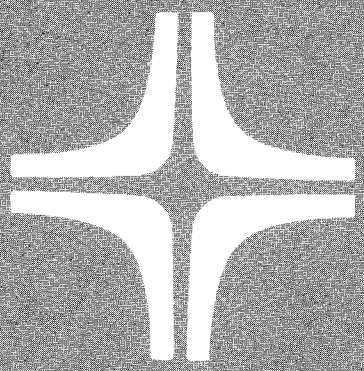
Copies of UP-9168 will be available for 6 months after the release of 7.1. Should you need additional copies of this edition, you should order them within 90 days of the release of 7.1. When ordering the previous edition of a manual, be sure to identify the exact revision and update packages desired.

Additional copies may be ordered by your local Sperry Univac representative.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ (less DE, GZ, HA) MZ, 18U, 19U, 20U, 21U, 28U, 29U, 75U and 76U	Mailing Lists DE,GZ,HA,18,19,20, 21,75 and 76 (Covers and 193 pages)	Library Memo for UP-9168 Rev. 1
		RELEASE DATE: September, 1981

BASIC

OS/3



Programmer Reference

SPERRY UNIVAC

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented at manual distribution time. To ensure that you have the latest information regarding levels of implementation and functional availability, please consult the appropriate release documentation or contact your local Sperry Univac representative.

Sperry Univac reserves the right to modify or revise the content of this document. No contractual obligation by Sperry Univac regarding level, scope, or timing of functional implementation is either expressed or implied in this document. It is further understood that in consideration of the receipt or purchase of this document, the recipient or purchaser agrees not to reproduce or copy it by any means whatsoever, nor to permit such action by others, for any purpose without prior written permission from Sperry Univac.

Sperry Univac is a division of the Sperry Corporation.

This document was prepared by Systems Publications using the SPERRY UNIVAC UTS 400 Text Editor. It was printed and distributed by the Customer Information Distribution Center (CIDC), 555 Henderson Rd., King of Prussia, Pa., 19406.

PRINTED IN U.S.A.

PAGE STATUS SUMMARY

ISSUE: Update B – UP-9168 Rev. 1
RELEASE LEVEL: 8.0 Forward

Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.
PSS	1	B
Preface	1, 2	Orig.
Contents	1 thru 5	Orig.
1	1, 2 3 4, 5 6 7, 8	Orig. B A Orig. B
2	1 thru 8 9 10 thru 12	Orig. A Orig.
3	1 thru 31 32 33 thru 68	Orig. A Orig.
4	1 thru 6 7 thru 9 10 thru 22	Orig. B Orig.
5	1, 2 3 4 thru 14 15 16	Orig. B Orig. B Orig.
6	1 thru 12 13 14 thru 18 19	Orig. B Orig. B
7	1 thru 3	Orig.
8	1 2 thru 4	B Orig.
Appendix A	1 thru 6	Orig.
Appendix B	1 2 3	Orig. B Orig.
Appendix C	1 thru 5 6 7 thru 12	Orig. B Orig.
Index	1 thru 10	Orig.
User Comment Sheet		

Part/Section	Page Number	Update Level

Part/Section	Page Number	Update Level

All the technical changes are denoted by an arrow (➔) in the margin. A downward pointing arrow (↓) next to a line indicates that technical changes begin at this line and continue until an upward pointing arrow (↑) is found. A horizontal arrow (➔) pointing to a line indicates a technical change in only that line. A horizontal arrow located between two consecutive lines indicates technical changes in both lines or deletions.

Preface

This reference manual describes the SPERRY UNIVAC Operating System/3 (OS/3) BASIC (Beginner's All-Purpose Symbolic Instruction Code) System, which permits the user to prepare, test, and execute programs while operating from a workstation. This manual is for the experienced BASIC programmer.

The organization of the manual is as follows:

- Section 1. System Description

  Provides the reader with a general overall knowledge of the components of the BASIC system.

- Section 2. Language Elements

  Discusses the elements that comprise the language used in constructing programs.

- Section 3. Source Language Statements

  Describes each BASIC source language statement according to category that is available to the user in constructing his BASIC program. The statements are presented alphabetically within each category.

- Section 4. File Support

  Describes the file-related statements and access methods supported under BASIC. The file-related statements are presented alphabetically.

- Section 5. BASIC Commands

  Describes each BASIC edit command available for preparing BASIC programs. These commands allow a user to name a program, execute a program, manipulate the source language statements in a program, and return control to the operating system. The BASIC commands are presented in alphabetical order.

- Section 6. BASIC Program Techniques

  Contains techniques used in constructing BASIC programs. These techniques include the hierarchy of arithmetic operations and the use of programming aids such as: lists, tables, matrixes, built-in functions, and multiline functions.

- Section 7. Errors and Debugging

  Describes the various user errors that may occur in preparing a BASIC program and the required correction facilities.

- Section 8. BASIC in a Batch Environment

  Contains programming techniques for BASIC in a batch environment. These techniques include items of special concern to the batch user, such as messages with a reply, source statement syntax errors, and running of a program.

- Appendix A. Summary of BASIC Statement and Command Formats with Examples

  Lists statement and command formats and descriptions. Examples are provided for each entry.

- Appendix B. Sample BASIC Session

  Shows a complete terminal session.

- Appendix C. BASIC Error Messages

  Contains a numerical list of OS/3 BASIC error messages.

Contents

PAGE STATUS SUMMARY

PREFACE

CONTENTS

1. SYSTEM DESCRIPTION

1. GENERAL 1-1

1.2. TERMINALS SUPPORTED BY BASIC 1-2

1.3. LOGON PROCEDURE 1-3

1.4. SOURCE PROGRAM CONSTRUCTION 1-3

1.5. BASIC SYNTAX CHECKER 1-4

1.6. BASIC COMMAND PROCESSOR 1-5
1.6.1. Program Execution 1-5
1.6.2. Program Listing 1-6
1.6.3. Saving a Program 1-7
1.6.4. File Organization of a Saved File 1-7
1.6.5. Using a Saved Program 1-7
1.6.6. Returning Control to the System 1-7
1.6.7. Deleting Program Lines 1-8
1.6.8. Pause User Program 1-8
1.6.9. Terminating BASIC 1-8

1.7. LOGOFF PROCEDURE 1-8

2. LANGUAGE ELEMENTS

2.1.	GENERAL		2-1
2.2.	CHARACTERS		2-1
2.3.	CONSTANTS		2-1
2.4.	VARIABLES		2-4
2.5.	EXPRESSIONS		2-6
2.6.	FUNCTION REFERENCES		2-7
2.7.	CHANNEL SETTER		2-11
2.8.	STATEMENTS		2-12

3. SOURCE LANGUAGE STATEMENTS

3.1.	INTRODUCTION		3-1
3.2.	DECLARATION STATEMENTS		3-3
3.2.1.	DEF Statement	(DEF)	3-4
3.2.2.	DIM Statement	(DIM)	3-6
3.2.3.	FNEND Statement	(FNEND)	3-8
3.3.	REMARK STATEMENT	(REM)	3-9
3.4.	ASSIGNMENT STATEMENT	(LET)	3-10
3.5.	CONTROL STATEMENTS		3-11
3.5.1.	END Statement	(END)	3-12
3.5.2.	FOR and NEXT Statements	(FOR and NEXT)	3-13
3.5.3.	GOSUB and RETURN Statements	(GOSUB and RETURN)	3-16
3.5.4.	GOTO Statement	(GOTO)	3-17
3.5.5.	IF Statement	(IF)	3-18
3.5.6.	ON Statement	(ON)	3-20
3.5.7.	PAUSE Statement	(PAUSE)	3-21
3.5.8.	STOP Statement	(STOP)	3-22
3.5.9.	RANDOMIZE Statement	(RANDOMIZE)	3-23
3.5.10.	TIME Statement	(TIME)	3-24
3.5.11.	SYSTEM Statement	(SYSTEM)	3-25
3.6.	DATA INPUT/OUTPUT STATEMENTS		3-25
3.6.1.	INPUT Statement	(INPUT)	3-26
3.6.2.	LINPUT Statement	(LINPUT)	3-27
3.6.3.	MARGIN Statement	(MARGIN)	3-28
3.6.4.	PRINT Statement	(PRINT)	3-29
3.6.5.	READ and DATA Statements	(READ and DATA)	3-33
3.6.6.	RESTORE and RESET Statements	(RESTORE and RESET)	3-35

3.6.7.	USING Statement	(USING)	3-36
3.6.7.1.	Formatting String Output		3-37
3.6.7.2.	Formatting Numeric Output		3-38
3.6.7.3.	Use with PRINT Statement		3-39
3.7.	MATRIX OPERATION STATEMENTS	(MAT)	3-42
3.7.1.	Matrix Dimensioning		3-44
3.7.2.	MAT Addition, Subtraction, and Multiplication Statements		3-45
3.7.3.	MAT Constant Statement		3-48
3.7.4.	MAT Identity Statement		3-49
3.7.5.	MAT INPUT Statement		3-50
3.7.6.	MAT Inversion Statement		3-51
3.7.7.	MAT LINPUT Statement		3-52
3.7.8.	MAT Null Statement		3-53
3.7.9.	MAT PRINT Statement		3-54
3.7.10.	MAT READ Statement		3-55
3.7.11.	MAT Scalar Multiply Statement		3-56
3.7.12.	MAT Transpose Statement		3-57
3.7.13.	MAT Vector Multiplication Statement		3-58
3.7.14.	MAT Zeros (0's) Statement		3-59
3.8.	PROGRAM SEGMENTATION		3-59
3.8.1.	CALL Statement	(CALL)	3-60
3.8.2.	CHAIN Statement	(CHAIN)	3-62
3.8.3.	LIBRARY Statement	(LIBRARY)	3-63
3.8.4.	SUB Statement	(SUB)	3-64
3.8.5.	SUBEND Statement	(SUBEND)	3-66
3.8.6.	SUBEXIT Statement	(SUBEXIT)	3-67
3.9.	CHANGE STATEMENT	(CHANGE)	3-68

4. FILE SUPPORT

4.1.	INTRODUCTION		4-1
4.2.	FILE DESCRIPTION		4-1
4.3.	FILE STATEMENTS		4-3
4.3.1.	FILE Statement	(FILE)	4-5
4.3.2.	INPUT Statement	(INPUT)	4-10
4.3.3.	LINPUT Statement	(LINPUT)	4-12
4.3.4.	MARGIN Statement	(MARGIN)	4-13
4.3.5.	Matrix I/O Statements		4-14
4.3.6.	PRINT Statement	(PRINT)	4-16
4.3.7.	READ Statement	(READ)	4-18
4.3.8.	RENAME Statement	(RENAME)	4-19
4.3.9.	RESET Statement	(RESET)	4-20
4.3.10.	SCRATCH Statement	(SCRATCH)	4-21
4.3.11.	WRITE Statement	(WRITE)	4-22

5. BASIC COMMANDS

5.1.	INTRODUCTION		5-1
5.1.1.	Definitions		5-1
5.2.	COMMANDS		5-2
5.2.1.	BYE	(BYE)	5-3
5.2.2.	DELETE	(DELETE)	5-4
5.2.3.	HELP	(HELP)	5-5
5.2.4.	LIST	(LIST)	5-6
5.2.5.	MERGE	(MERGE)	5-7
5.2.6.	MODIFY	(MODIFY)	5-8
5.2.7.	NEW	(NEW)	5-9
5.2.8.	OLD	(OLD)	5-10
5.2.9.	PRINT	(PRINT)	5-11
5.2.10.	RESEQUENCE	(RESEQUENCE)	5-12
5.2.11.	RUN	(RUN)	5-13
5.2.12.	RUNOLD	(RUNOLD)	5-14
5.2.13.	SAVE	(SAVE)	5-15
5.2.14.	SYSTEM	(SYSTEM)	5-16

6. BASIC PROGRAM TECHNIQUES

6.1.	INTRODUCTION	6-1
6.2.	HIERARCHY OF ARITHMETIC OPERATIONS	6-1
6.3.	USE OF LOOPS	6-3
6.4.	USE OF LISTS AND TABLES	6-5
6.5.	USE OF BUILT-IN FUNCTIONS	6-7
6.5.1.	Mathematical Functions	6-7
6.5.2.	Specialized Functions	6-8
6.5.3.	String Functions	6-12
6.5.4.	File Functions	6-13
6.6.	USE OF MULTILINE FUNCTIONS	6-15
6.7.	USE OF SUBPROGRAMS	6-16
6.8.	USE OF FILES	6-17
6.9.	HINTS FOR MORE EFFICIENT CODE	6-19

7. ERRORS AND DEBUGGING

7.1.	GENERAL	7-1
7.2.	ERRORS PREVENTING RUNNING OF PROGRAM	7-1
7.3.	LOGIC ERRORS	7-2

8. BASIC IN A BATCH ENVIRONMENT

8.1. INTRODUCTION 8-1

8.2. PROGRAMMING CONSIDERATIONS 8-1
8.2.1. BASIC Messages with a Reply 8-1
8.2.2. BASIC Commands or Source Statements 8-1
8.2.3. Syntax Errors in Source Statements 8-2
8.2.4. RU Command 8-2

8.3. BASIC BACKGROUND OPERATION 8-3

APPENDIXES

A. SUMMARY OF BASIC STATEMENT AND COMMAND FORMATS

B. SAMPLE BASIC SESSION

C. BASIC ERROR MESSAGES

INDEX

USER COMMENT SHEET

FIGURES

1-1. BASIC System Overview 1-2

8-1. BASIC Batch Environment Printout 8-4

B-1. Sample BASIC Session B-1

TABLES

2-1. List of Mnemonics 2-10

3-1. List of BASIC Statements 3-1
3-2. Relation Symbols 3-18

4-1. BASIC File Statements 4-3

6-1. Nested Loops 6-5

A-1. BASIC Statement and Command Formats A-2

1. System Description

1.1. GENERAL

The SPERRY UNIVAC Operating System (OS/3) BASIC (Beginner's All-Purpose Symbolic Instruction Code) System provides the workstation user with the capability of generating, modifying, and executing programs written in the BASIC language. The BASIC system also provides the user with the capability of saving his programs on direct-access storage for subsequent processing and updating.

The BASIC language is an interactive programming language designed to be easy to use, yet meet the requirements of both business and scientific programming. The BASIC language available on the OS/3 operating system complies with the *American National Standard Minimal BASIC, X3.60-1978* and includes Dartmouth features and compatibility. It provides a powerful, yet simple set of commands allowing the novice to learn the language quickly, and yet gives the experienced programmer an extensive list of features for various applications.

Figure 1-1 shows an overview of the BASIC system. After logging on, the BASIC system is invoked by typing in the BASIC command at the workstation. The system then loads the BASIC compiler, responds with READY, and the user begins to construct or modify his source program. Each BASIC statement entered is analyzed by a syntax checker immediately for syntax errors such as invalid constants, expressions, and construction. If an error is detected, BASIC prints a question mark (?) and the statement in error, up to the first character where the error occurred. The user may then correct the error and proceed to the next statement.

After the user has completed his program or part of a program, he may issue the RUN command to compile and execute the sequence of statements. The BASIC compiler performs a second syntax check for global errors during object code generation. These errors are detected when the source program is analyzed in its entirety rather than on an individual source line basis. Examples are illegal nesting, undefined function references, and illegal line-number references (1.6.1). If an error is detected, BASIC prints the line number of the source statement in error and an appropriate diagnostic message (Section 7 and Appendix C).

After compilation and execution of the program, the results are returned to the user's terminal. The user may then use the SAVE command to save a copy of the current program in a library file. The program is stored using the program name supplied by the user.

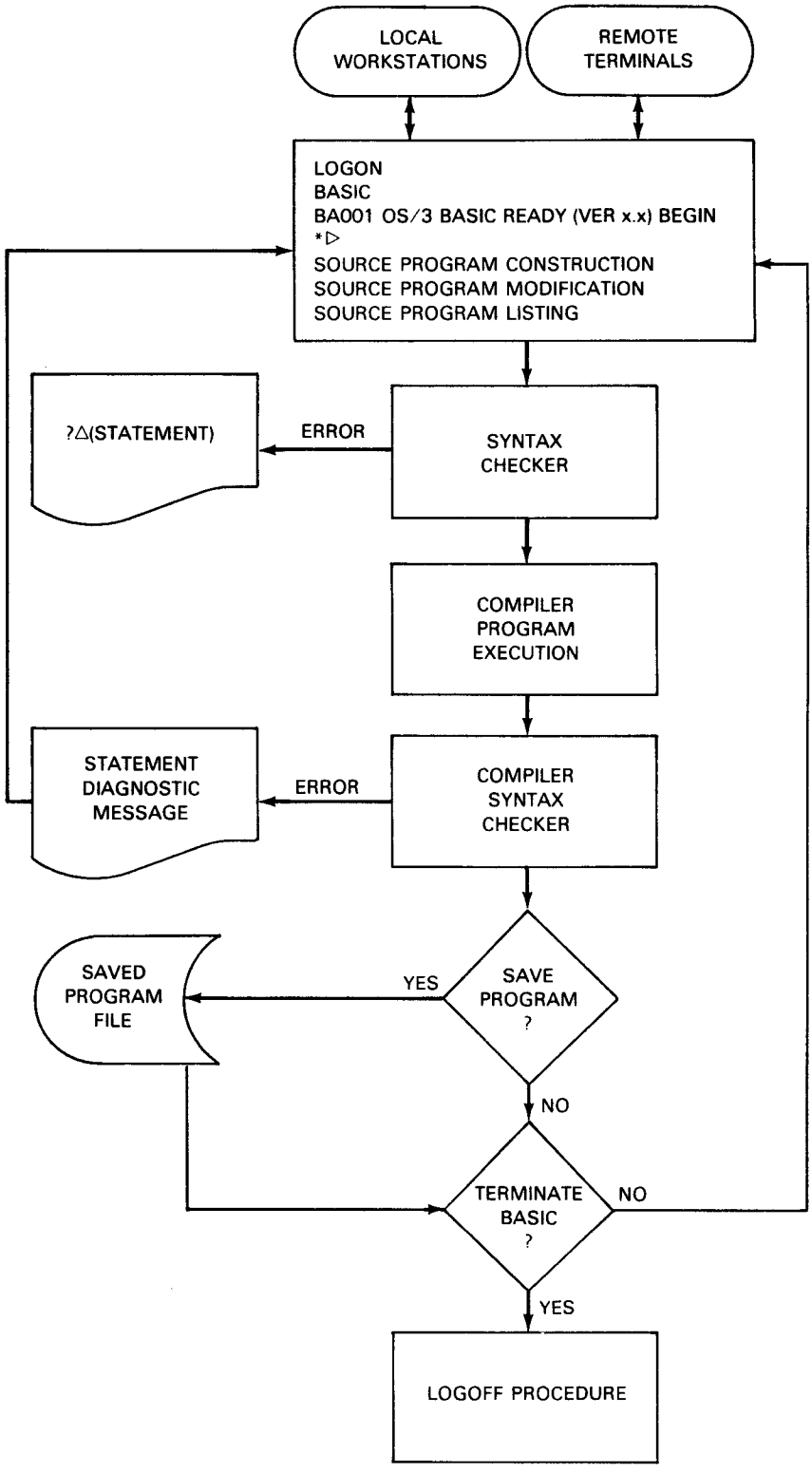


Figure 1—1. BASIC System Overview

1.2. TERMINALS SUPPORTED BY BASIC

The BASIC system supports local and remote terminals. In this manual, local units are referred to as workstations, and remote units are referred to as terminals.

1.3. LOGON PROCEDURE

To initially log on to the operating system, the user must enter the LOGON command in the system mode from his terminal. This command identifies the user to the operating system.

Format:

```
LOGON user-id [,acct][,password]
```

where:

LOGON
: Specifies that the user wants to log on.

user-id
: Is a 1- to 6-character alphanumeric code you enter to identify yourself to the system. The user-id is used by the system to correctly route messages, job and command output, and to determine which commands you may use on the system. *The user-id must begin with an alphabetic character.*

acct
: Is a 1- to 4-character alphanumeric code that is used for system time accounting.

password
: Is a 1- to 6-character alphanumeric code that controls your access to the overall system.

For a more detailed discussion of the LOGON command, refer to the OS/3 workstation user guide, UP-8845 (current version).

1.4. SOURCE PROGRAM CONSTRUCTION

In system mode, the user invokes BASIC by issuing the following executive command:

```
BASIC
```

Control is transferred to BASIC, which immediately prints the following message:

```
BA001 OS/3 BASIC READY (VER x.x) BEGIN
```

At this time, the user is at the command level in BASIC. If a command other than NEW or OLD is entered, the syntax checker is called immediately to process the user's first source statement.

After the compiler is called, the system responds with an asterisk, which requests source input. A line of input consists of a single BASIC source language or a BASIC editing command, followed by the *TRANSMIT* function. If an all-blank line is transmitted, BASIC simply reissues the prompt. The BASIC source language statements and editing commands are described in detail in Sections 3, 4, and 5. Input lines may not be continued beyond one terminal line.

BASIC distinguishes program source statements from editing commands by requiring that the source statements be prefixed by a line number. A line number consists of 1 to 5 digits with a value between 1 and 99999. Line numbers are used to determine the logical sequence of statements. In a BASIC program file, lines of source text may be entered in an arbitrary sequence.

The lines of source text are processed by the BASIC syntax checker and syntactically correct statements are added in source form to the user's program file. This program file, which is built up in the user's workspace, is not saved unless the user issues a SAVE or RESEQUENCE command. Statements entered at the terminal, which have a syntax error, initiate diagnostics and are not added to the user's program file.

BASIC editing commands are executed immediately and are not included in the user's program file. The user's program file is compiled and executed when the RUN command is issued.

After a line of input is processed, the system responds with an asterisk on a new line requesting another line of input from the terminal.

The maximum acceptable input line is 80 characters.

Because source statements are cataloged by line number in a user's program file, no more than one statement can have the same specific line number. Therefore:

1. If the line number of a syntactically correct source statement matches the line number of a statement in the current user's program file, the new statement replaces the old statement.

2. A null statement such as 140 followed by the TRANSMIT key deletes a statement with a matching line number in the current user's program file.

NOTE:

The workstation screen clears if it is sitting idle for a period of time. To return to the original screen, press function key 19 (FK19) or function/workstation mode keys simultaneously. Do not press the MESSAGE WAITING key.

Section 6 describes the techniques that a BASIC user can employ in constructing his program. Techniques for formatting formulas, using loops, formatting lists and tables, and using specialized functions are described with appropriate examples.

1.5. BASIC SYNTAX CHECKER

The BASIC syntax checker analyzes single BASIC source language statements. If a syntax error is detected, the system responds with a question mark (?) followed by a copy of the incorrect statement up to the first character in error. The user may then retype the remainder of the source statement followed by the TRANSMIT key as the next line of input.

Example:

The user types in the following line and then presses the TRANSMIT key:

```
24 IF A=B THEN GOTO 41
```

The system responds with:

```
?24 IF A=B THEN
```

because the GOTO following THEN is incorrect.

The user may then type in the following and then press the TRANSMIT key:

```
41
```

BASIC processes the following completed statement:

```
24 IF A=B THEN 41
```

The following types of errors are detected by the BASIC syntax checker:

- Incorrect constants, identifiers, function names, line numbers, and statement verbs
- Incorrect expressions caused by unbalanced parentheses, implicit multiplication, and illegal operand-operator-operand sequences (e.g., two operators together as in A*-B)
- Incorrect statement construction, such as no THEN clause following IF

Global syntax errors (e.g., transfer to a line number not included in a program) are detected by the BASIC compiler.

Lines of input which are not prefixed by a line number automatically bypass the syntax checker and are treated as commands. The BASIC command processor responds with a question mark (?) to an invalid command, which frequently results from typing a source statement without its line number.

If the error in a rejected BASIC statement is not obvious, the user may issue a HELP command. This will result in a short explanation of the error being displayed at the terminal. Corrective action is often suggested by the explanation.

When errors are detected by the syntax checker, only the portion of the statement which is correct will be displayed at the terminal. The user should complete the statement and retransmit it to BASIC. In the case where the user does not want to correct the statement, but wants to enter a new statement or a command, he should back up the cursor to the start-of-entry symbol (▷) and erase the line. A new statement may now be entered.

1.6. BASIC COMMAND PROCESSOR

The BASIC system provides a set of edit commands which are described in detail in Section 5. The editing commands are integrated with the BASIC source language statements so that the user does not have to manually switch between edit and program construction modes.

1.6.1. Program Execution

The BASIC compiler is a one-pass, load-and-go system. The compiler generates object code that provides for program execution following the statement. The RUN command instructs the BASIC system to compile and execute the sequence of statements currently contained in the user's program file. This sequence of statements need not constitute a logically complete BASIC program, because the compiler automatically generates code to terminate program execution following the last statement. The last statement in a program file must always be an END statement, whether or not the program is logically complete.

In addition, the BASIC compiler does extensive global syntax checking. Each syntax error results in a message to the user's terminal consisting of the line number of the source statement that caused the error and an appropriate diagnostic.

Example:

```
BA039 INCORRECT NESTING OF FOR-NEXT STATEMENT
BA027 LOADER ERROR AT LINE 00020
*
```

As the program is loaded, a diagnostic is displayed for each error encountered; if errors are detected, the user is returned to the syntax checker. If no compiler errors are detected, the object code is automatically executed. The following types of global syntax errors are detected by the BASIC compiler syntax checker:

- Overflow and underflow resulting from conversion of numeric constants to floating-point internal representation
- Reference to an undefined function and redefinition of a defined function
- References to nonexistent or invalid line number (e.g., GOTO, GOSUB, IF-THEN, ON)
- NEXT before FOR, or no NEXT matching a FOR
- Illegal nesting of FORs with same index
- Illegal nesting of FORs with different indexes
- Statements leading to unpredictable results
- Duplicate parameters in a function definition
- Illegal DEF-FNEND statement ordering

If an OLD program is being executed, and there are statements which were flagged by the syntax checker but have not yet been corrected, the loader will display an error message:

```
BA026 UNCORRECTED ERROR IN SOURCE
BA027 LOADER ERROR AT LINE 00760
```

The user should go back and correct the lines in error before attempting to RUN the program again.

The code generated by the BASIC compiler includes tests for a number of run-time errors. Each run-time error results in a type-out to the user's terminal consisting of the source statement that resulted in the error and an appropriate diagnostic.

Example:

```
BA015 ARRAY SUBSCRIPT OUT OF RANGE
BA062 EXECUTION STOPPED AT LINE 00230
*▷
```

Program execution terminates automatically when a run-time error is detected. See Appendix C for a complete list of diagnostics.

1.6.2. Program Listing

The LIST or PRINT command can be used to display all or part of a program at the user's terminal.

Example:

```
LIST 150 - 175
```

Only those lines numbered 150 to 175, inclusive, are listed. Lines of source text are listed as they were typed in.

1.6.3. Saving a Program

The SAVE command can be used to save a copy of the user's current program file in an OS/3 library file. The file name, supplied by the user, is used to locate the file on the disk. The program name is used for an element name within the library. The program is saved in source statement form. If a BASIC program with the same program name has been previously saved on the user's disk file, the system will respond:

```
IS100 FILE/MODULE ALREADY EXISTS;OK TO WRITE IT? (Y,N)
```

If the user responds with Y or YES, the current program file will replace the previously saved program. For responses with an N or NO, the control will be returned to the user without overwriting the previously saved program.

The message is repeated for a response different from Y, YES, N, or NO. For an example of saving a program, refer to the SAVE command description in Section 5.

1.6.4. File Organization of a Saved File

All files saved by BASIC, or OLD programs recalled by BASIC, are stored in standard OS/3 library files. The user is required to supply at least the program and file names. BASIC will check the system catalog to see if it lists the file. If it does, the file password, if any, will be verified and the volume name listed in the catalog will be used. If the file is not listed in the catalog, the user will be required to supply a volume name.

When the user invokes the OLD command, all lines of source are processed by the syntax checker. If a syntax error is discovered while reading a statement from the source file, the line is written to the terminal, preceded by a question mark, and rejected. It will then be entered into the work file with a notation that the line must be corrected before the program may be run. The user must wait until the entire file is read before he can enter lines from the terminal. BASIC will respond with an * when it is ready.

Programs saved by BASIC may be listed or punched using the OS/3 utility LIBS.

1.6.5. Using a Saved Program

The OLD command can be used to load a program saved on the user's OS/3 library file into his workspace. When the OLD command is issued, the user must also supply the file information of one of the BASIC programs saved in a library file. The saved program then becomes his active program file. The copy of the program on disk is unchanged.

The OLD and NEW commands may be issued at any time during a BASIC session. In either case, the current contents of the user's active program file are lost and the file is renamed.

Another command, RUNOLD, allows the user to quickly execute a saved program without the overhead of copying the source and compiled object code to the workspace.

1.6.6. Returning Control to the System

During the BASIC session, it may be necessary for the user to return control to the system, so that certain system commands such as FSTATUS, ASK, etc, may be issued. In order to facilitate returning control to the system, BASIC provides the user with the SYSTEM command. The SYSTEM command causes BASIC to interrupt to the system, and the user can subsequently return to BASIC by issuing the RESUME command.

1.6.7. Deleting Program Lines

Basic statements that have been stored in the work file may be removed by typing their line number, as explained previously. A command is also available to remove several lines with a single command.

Example:

```
DELETE 126-129, 500
```

1.6.8. Pause User Program

Function key 1 may be used to pause the execution of the user program. When the terminal operator inputs function key 1 during the execution of the user program, BASIC stops execution and the following message is displayed on the terminal:

```
BA063 EXECUTION PAUSED AT LINE xxxxx CONTINUE (Y,N)?
```

If the user responds with Y, the user program is continued. If the user responds with N, the user program is terminated. If the user responds with other than Y or N, the message is repeated. To pause program execution when BASIC is requesting input, the user inputs the requested data, inputs function key 1, and ends with TRANSMIT. BASIC then prints the pause message. To pause program execution during output, the entire workstation screen must be full of data. At the end of the last line on the screen, the user inputs function key 1 and then inputs function key 19. The pause message is printed on the first line of the newly cleared screen.

1.6.9. Terminating BASIC

The BYE command terminates the BASIC session, and all storage space occupied by the program is released to the system. BASIC issues a warning to the user prior to executing program termination. When the BYE command is used, BASIC determines if the user program has been saved in the library file. If it was not saved, the following message will be displayed on the terminal:

```
BA118 SOURCE MODULE NOT SAVED - TERMINATE (Y,N)?
```

If the user responds with Y, the following message will be displayed and the BASIC task will terminate.

```
BA113 BASIC TASK NORMAL TERMINATION
```

If the user responds with N, the BYE command will be rejected, allowing the user to save the program. The message is reissued for any other response.

1.7. LOGOFF PROCEDURE

The LOGOFF command terminates the user session. This command must be the last issued in the task in the following format:

```
LOGOFF
```

2. Language Elements

2.1. GENERAL

The BASIC language is made up of elements that can be combined in various ways to construct programs and subroutines. The language elements are divided into the following categories:

- Characters
- Constants
- Variables
- Expressions
- Function references
- Channel setter
- Statements

2.2. CHARACTERS

BASIC programs are constructed from a set of 58 distinct characters. A character is defined as a letter, digit, delimiter, or special character.

Letter:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digit:	0 1 2 3 4 5 6 7 8 9
Delimiter:	Operator or separator
	Operator: + - * / ( ) < > & ↑
	Separator: , . ; △ " :
Special Character:	$ @ # ? % '

In addition, BASIC programs use open-string and string characters.

open-string character	letter, digit, operator, special character, period (.), semicolon (;), and double quote (").
closed-string character	letter, digit, operator, special character, comma (,), period (.), a semicolon (;), or a blank (△).

Note the following conventions:

1. Blanks:

   The character blank is designated in the syntax by the symbol △. Any spaces that appear in the syntax equations do not denote blanks in the BASIC language. Blanks are only significant in BASIC when they appear in a comment or a string constant.

2. Quote:

   The character quote (") is used to delimit the beginning and end of a closed-string constant. If a quote is required within a closed string, use two consecutive quotes.

3. Asterisk:

   Exponentiation is specified by a pair of asterisks (**). A vertical arrow (↑) is also permitted, where applicable.

2.3. CONSTANTS

Constants are used to specify data values. There are three types of constants: decimal numbers, string constants, and line numbers.

- Decimal number

  A fraction that may be optionally followed by an exponent field. A fraction is defined as a series of 1 or more digits that may contain an optional decimal point. The decimal point may precede, follow, or be embedded in the series of digits. The exponent field indicates the power of 10 that the fraction is to be multiplied by and consists of the letter E followed by an optional sign and 1 or 2 digits. The sign may be + or - and, if omitted, is assumed to be +.

  Examples:

  Fraction: `9, 9., .9, 9.9`

  Exponent: `E1, E+1, E+01, E-1, E-01`

- String constant:
  - Closed string:

    A quote followed by a series of 0 to 4095 string characters followed by a quote

    Example:

    ''A△B'' or ''BILL'' ''S''

  - Open string:

    A series of 1 to 4095 open-string characters or blanks or quotes

    Example:

    A△B

- Line number:

  A series of one to five digits without any sign, decimal point, or exponent field. It must be in the range of 1 to 99999.

Note the following conventions:

1. Decimal numbers:

   All decimal numbers are converted and stored internally in floating-point format. The exponent occupies 7 bits and indicates the power to which the number 16 must be raised. The sign occupies one bit. In floating-point format, the mantissa occupies 24 bits and contains a 6-digit hexadecimal number in normalized form. In BASIC, if the value of the fraction part of a decimal number, disregarding the decimal point, exceeds $2^{24}-1$, the number is rounded and trailing digits are lost.

   Example:

   12.3456789

   This decimal number is acceptable, but is (effectively) rounded to the following:

   12.345679

   If the mantissa is nonzero, the magnitude of the floating-point number has the following range:

   $16^{-65} \leq M < 16^{63}$ (approximately $10^{-78} \leq M < 10^{75}$)

   Overflow and underflow conditions for numeric constants are processed as errors.

2. String constants:

   All string constants are stored in EBCDIC code. A 2-byte length field is prefixed to each string before it is stored; the value of the length byte is not included. If a given string constant contains more than 4095 characters, it is truncated at the right. Note that an open-string constant, as opposed to a closed-string constant, cannot contain a comma. Moreover, an open-string constant is permitted only as input to the READ and INPUT statements. Note that it is not possible to enter a string constant in a program longer than 74 characters, because the maximum line length is 80 characters.

   Within a closed-string constant, two consecutive quotes are interpreted as a single quote.

3. Line numbers:

   A line number that is an integer between 1 and 99999 must precede each statement in a BASIC program. The line numbers specify the logical sequence of statements in a program (ascending order). They are also used as statement labels for transferring control during program execution.

   Leading zeros in a line number are ignored in the sense that 00175 is equivalent to 175.

2.4. VARIABLES

Variables are used to designate arbitrary data values of a fixed type. In BASIC, the user may construct scalar variables and array variables. A scalar variable is defined as a numeric variable or a string variable. An array variable is defined as a numeric array or string array. A numeric reference may be a numeric variable or a numeric array. A string reference may be a string variable or a string array.

- Scalar variable:

  A numeric variable or string variable

  - Numeric variable:

    A letter optionally followed by a single digit

    Examples:

    ```
    X, X2
    ```

  - String variable:

    A letter followed by a dollar sign ($), or a letter followed by a single digit, followed by a dollar sign.

    Examples:

    ```
    A$, J$, Q6$
    ```

- Array variable:

  A numeric array variable or string array variable

  - Numeric array variable:

    A letter followed by one or two subscript expressions enclosed in parentheses

    Examples:

    ```
    X(4), X(4,20), X(A+B)
    ```

  - String array variable:

    A letter followed by a dollar sign ($) followed by one or two subscripts enclosed in parentheses

    Examples:

    ```
    C$(20), C$(A+B), D$(A,C)
    ```

Note the following conventions:

1. Numeric variables:

   Numeric variables may only be assigned decimal numeric values.

2. Numeric array variables:

   Numeric array variables may only be assigned decimal numeric values.

   The upper bounds for a 1-dimensional or 2-dimensional numeric array may be explicitly specified by a dimension (DIM) statement (Section 3). An implicit upper bound of 10 for either dimension is implied if not specified. In either case, the lower bound is always 0.

3. Subscripts:

   A subscript may be defined using any arithmetic expression. During execution, the value used to locate the array element referenced is computed by rounding the subscript expression to the nearest integer. If the subscript value is not within the bounds specified (or implied) for that dimension of the referenced array, then the user is given an error message and program execution terminates.

   Two-dimensional numeric arrays are stored in row-major order.

4. String variables:

   String variables may only be assigned character string values. All such variables are initialized to the null string (zero length). A string variable may contain up to 4095 characters.

5. String array variables:

   String array variables may only be assigned character string values. All elements of these string array variables are initialized to the null string (zero length).

   The rules for numeric array variables regarding bounds and subscript evaluation apply to string array variables as well.

2.5. EXPRESSIONS

The expression is the BASIC facility for performing operations on data values. BASIC provides for both arithmetic numeric expressions and string expressions. Arithmetic numeric expressions specify arithmetic calculations; string expressions identify input/output. Unless otherwise stated, all expressions are assumed to designate single values.

- Arithmetic expression:

  A term optionally preceded by a minus (–) or plus (+) sign, or an arithmetic expression plus (+) or minus (–) a term

  Example:

  ```
  A**2*B-3
  ```

  - Term

    A factor or a term multiplied (*) or divided (/) by a factor

    Example:

    ```
    A**2*B
    ```

  - Factor

    A primary or a factor raised to a power (**) designated by a primary

    Example:

    ```
    A**2
    ```

  - Primary

    A decimal number, numeric reference, function reference, or an arithmetic expression enclosed in parentheses

    Example:

    ```
    2,A,SQR(X),(C-D)
    ```

- String expression:

  A string primary or a string expression followed by an ampersand (&) denoting concatenation, followed by another string expression

  Example:

  ```
  ''ABC''&B$
  ```

- String primary

  A closed-string reference or function reference

  Example:

```
A$,SEG$(D$,6,8),''AB''
```

Note the following conventions:

1. Mixed mode expressions are treated as errors.
2. The exponentiation operator (**) may be written as a vertical arrow (↑), where applicable.
3. A**B**C is compiled as (A**B)**C.
4. Parentheses may be used to factor subexpressions.
5. Overflow and underflow conditions existing during the evaluation of arithmetic expressions are treated as errors.
6. Division by zero is treated as an error.
7. Zero to a negative power is treated as an error.
8. A negative number can be raised only to a nonzero positive integer number. The maximum value of the positive integer is 15. Any violation of this rule is treated as an error.

2.6. FUNCTION REFERENCES

An expression may contain references to the following types of functions:

- Specific built-in functions provided within the BASIC system
- User-defined numeric and string functions

Function references consist of a function name, followed by an argument (list) enclosed in parentheses. All built-in functions have between zero and three arguments. In each case, the arguments are evaluated and control is tranferred to an out-of-line routine for evaluating the referenced function.

The resulting (single) numeric or string value replaces the function reference in the containing expression.

- Built-in function

  A function name optionally followed by an expression or list of expressions enclosed in parentheses

- Function name

  ABS, ATN, CHR$, CLK$, COS, COT, DAT$, DET, EBC, EXT, INT, LEN, LOC, LOF, LOG, MAR, MOD, NUM, PER, POS, RND, SEG$, SGN, SIN, STR$, SQR, TAN, TIM, TYP, USR$, VAL ←

- User-defined function

  FN followed by a letter and optional dollar sign, followed by an argument list enclosed in parentheses

  Example:

```
FNC$(C$,Z)
```

- Argument list

  Expression optionally followed by up to 15 expressions. A comma is used to separate one expression from another

  Example:

```
A,B$,''ABC''&''DEF''...
```

Note the following conventions:

1. SIN(x), COS(x), TAN(x), COT(x), and ATN(y) designate the functions sine, cosine, tangent, cotangent, and arctangent, respectively, and the argument x and the result of ATN are angles measured in radians.

2. EXP(x) designates exponentiation, e^x. Overflow occurs if x is too large ($x > 174.6$).

3. LOG(x) designates the natural logarithm of x. The LOG of zero or a negative number is treated as an error.

4. ABS(x) designates the absolute value of x, $|x|$.

5. SQR(x) designates the square root of x. A negative argument is treated as an error.

6. RND(x) designates a pseudorandom number as follows:

   a. If $x > 0$, then RND(x) is a function of x whose value is in the open interval (0,1).

   b. If $x < 0$, the system supplies an arbitrary random number in open interval (0,1).

   c. If $x = 0$, the system supplies a pseudorandom number that is a function of the previous random number generated by RND. If $x = 0$, the first time RND is called in a program, the system will supply a fixed number in the open interval (0, 1).

   d. If no argument is used, $x = 0$ is assumed. To generate a sequence of pseudorandom numbers, the user would call any of these options followed by repeated calls to option c. With this option, the RANDOMIZE statement should be used to generate a unique sequence of random numbers.

7. INT(x) designates the largest integer not exceeding x.

   Example:

```
INT(2.985) = 2.          INT(-2.015) = -3.
```

8. SGN(x) designates the sign of x.

$$SGN(x) = \begin{cases} +1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0 \end{cases}$$

9. FNA to FNZ designates one of the 26 user-defined numeric functions and FNA$ to FNZ$, one of 26 user-defined string functions (see the DEF statement).

10. DET is a pseudofunction and may be used to obtain the value of the determinant of the last matrix inverted.

11. LEN (X$) computes the length, in characters, of the string X$. This will be a value between 0 and 4095.

12. MOD (x,y) is the modulus remainder of x divided by y: (x-y * INT (x/y))

13. POS (A$, B$, X) determines the location in string A$ of the first character of the first occurrence of the string B$ beginning at or after position X in A$. This will return zero if B$ does not occur in A$.

14. TIM is the elapsed running time of the program in seconds, accurate to the nearest millisecond.

15. VAL (A$) returns the value of the number whose decimal representation is in string A$.

16. EBC (string) is a special function that takes a string of from one to three characters in length. It returns a value of the EBCDIC code for its argument. The argument is a character, or a 2- or 3-letter mnemonic for a character (e.g., EBC (ETX) = 3). See Table 2-1 for a list of mnemonics.

17. CHR$(x) returns a 1-character string consisting of the EBCDIC character with the code MOD (INT (x), 256). For example:

```
CHR$(193) = A.
```

    This function does not apply to device control characters. BASIC converts to blanks (X'40') any character with a hexadecimal value of X '00' to X'3F'.

18. CLK$ gives the time of day as an 8-character string in the form hh:mm:ss.

19. DAT$ gives the current date as an 8-character string in the form mm/dd/yy.

20. SEG$ (A$,x,y) locates the substring of A$ consisting of all characters between positions X and Y inclusive, and returns that string. An empty string is returned if X > Y, and the appropriate beginning or end of A$ is taken for X <=0 or Y>LEN (A$).

21. STR$ (X) converts X to its decimal representation as a string result.

22. USR$ is a 6-character string giving the user's *logon identifier* from the LOGON command.

23. LOC (#N) returns the current location of the file pointer for the file assigned to channel number N.

24. LOF (#N) returns the current end-of-file value (length of file) for the file currently assigned to channel number N.

25. MAR (#N) returns the current margin size for the file currently assigned to file number N.

Table 2—1. List of Mnemonics

Mnemonic	Value	Mnemonic	Value
ACK	46	LCJ	145
BEL	47	LCK	146
BS	22	LCL	147
CAN	24	LCM	148
CR	13	LCN	149
DC1	17	LCO	150
DC2	18	LCP	151
DC3	19	LCQ	152
DC4	60	LCR	153
DEL	7	LCS	162
DLE	16	LCT	163
DS	32	LCU	164
EM	25	LCV	165
ENQ	45	LCW	166
EOT	55	LCX	167
ESC	39	LCY	168
ETB	38	LCZ	169
ETX	3	LF	37
FF	12	NAK	61
FS	28	NUL	0
GS	29	RS	30
HT	5	SI	15
LCA	129	SO	14
LCB	130	SOH	1
LCC	131	SOS	33
LCD	132	SP	64
LCE	133	STX	2
LCF	134	SUB	63
LCG	135	SYN	50
LCH	136	US	31
LCI	137	VT	11

26. PER (#N,A$) returns the value +1 if the operation specified by A$ is valid for channel number N, 0 if the operation is invalid, and -1 if A$ does not specify one of the operations: INPUT, LINPUT, PRINT, READ, RENAME, RESET, SCRATCH, or WRITE. Operations may be invalid if they are applied to an unopened file or if the user has restricted access to the file. INPUT, LINPUT, and READ are invalid if the file is empty or the current pointer is at end-of-file (LOC=LOF). A value of +1 returned by PER ensures that the specified operation will be allowed if it is the next operation issued against the file.

27. TYP (#N,A$) returns +1 if the file given by N currently has the type specified by A$, 0 if not, and -1 if A$ does not specify one of the operations: ANY, LIBRARY, NUMERIC, PERM, RANDOM, STRING, TERMINAL, TTY, or WORK. The terminal has type TTY, a scratch file has type WORK, an OS/3 library file has type LIBRARY, and an OS/3 MIRAM file has type PERM. Any open file has type ANY. NUMERIC and STRING are provided for compatibility and will always return a value of +1. A TERMINAL file is a sequential file for which the operations INPUT, LINPUT, and PRINT are valid. A RANDOM file is one for which the operations READ, WRITE, and RESET are valid. Currently all BASIC files have both type TERMINAL and type RANDOM except for the workstation file, which has type TERMINAL.

28. NUM returns the number of values input for the last vector MAT INPUT statement. If the vector has a trimmer, NUM is not updated.

2.7. CHANNEL SETTER

The channel setter is used in file-related statements to specify which data file is to be selected.

Format:

```
# expression
```

where:

`#`
: Identifies the channel setter.

`expression`
: Is a numeric expression that is evaluated at execution time.

Programming Notes:

1. The expression is truncated to an integer. The resultant value must be in the range 0 to 4095.
2. A channel setter of zero, or an omitted channel setter, selects the terminal.

Examples:

```
#3, #1, #3-J
```

2.8. STATEMENTS

The statement is the smallest complete unit of information in the BASIC system. Statements may be entered into a program, reordered, and executed.

There are two general classes of statements in BASIC: executable and nonexecutable. Executable statements designate particular actions to be performed; nonexecutable statements specify supplementary information.

Statement:

A line number followed by an executable statement or a nonexecutable statement

Executable statement:

Assign, control, input-output, matrix, and data file statements

Nonexecutable statement:

Declaration or remark statement

Each BASIC statement entered into a program must be prefixed with a line number. These line numbers determine the logical order of statements within a program. They are also used in several of the control statements to effect transfers of control.

Comments may be appended to any BASIC statement by prefixing the comment with an apostrophe ('). When the syntax checker scans a source statement, any characters after the apostrophe are ignored (except when the apostrophe is part of a string constant).

Each BASIC statement is described in detail in Section 3.

3. Source Language Statements

3.1. INTRODUCTION

This section describes the BASIC source language statements that are used in constructing a BASIC program. Each statement is described in detail with examples showing the use of each statement.

The BASIC source language statements are classified as either executable or nonexecutable. The statements are categorized as: declaration, remark, assignment, control, data input/output, matrix operations, program segmentation, change, and file support. Statements within these categories are presented alphabetically. Table 3-1 shows the list of all the BASIC source language statements.

Table 3—1. List of BASIC Statements (Part 1 of 2)

Statement Category	Statements	
	Executable	Nonexecutable
Declaration		DEF DIM FNEND
Remark		REM
Assignment	LET	
Control	FOR and NEXT GOSUB and RETURN GOTO IF ON STOP, PAUSE, and END SYSTEM	TIME
Data Input/Output	INPUT LINPUT MARGIN PRINT READ RESTORE and RESET USING	DATA

Table 3—1. List of BASIC Statements (Part 2 of 2)

Statement Category	Statements	
	Executable	Nonexecutable
Matrix Operations	MAT add, subtract, multiply MAT constant MAT identity MAT INPUT MAT inversion MAT LINPUT MAT null MAT PRINT MAT READ MAT scalar multiply MAT transpose MAT vector multiplication MAT zeros	
Program Segmentation	CALL CHAIN SUBEND SUBEXIT	LIBRARY SUB
Change	CHANGE	
File Support	FILE INPUT LINPUT MARGIN MATRIX I/O PRINT and USING READ RENAME RESET SCRATCH WRITE	

A BASIC program consists of any sequence of BASIC statements; each statement must be preceded by a line number and must be written on a single line of terminal input. The maximum number of statements in a program depends on the complexity of individual statements in a particular program. This limit is usually a function of the amount of main storage available to load the program, and is not a limit imposed by the compiler.

In describing the statements, the following conventions are used:

1. Keywords that may be used in the statement are in capital letters.
2. Names constructed using lowercase letters and embedded hyphens designate syntactic variables.
3. Brackets, [ ], are used to enclose optional parameters.
4. Braces, { }, are used to enclose alternatives.

5. Ellipsis, . . ., following an operand parameter indicates that the user may specify more than one parameter of that type.

   Example:

   ```
   READ variable-1 [,variable-2...]
   ```

   allows the READ statement to contain one or many input variables in the READ list.

   ```
   READ A
   READ A,B
   READ A,B,C
   ```

3.2. DECLARATION STATEMENTS

The declaration statements (DEF, DIM, and FNEND) explicitly specify the dimensions of arrays and define any defined functions that are referenced in a program.

DEF

3.2.1. DEF Statement

In addition to the built-in functions, the BASIC user can define other functions via the DEF statement.

Format:

```
DEF FN letter [$][(param-list)][local-list][=expression]
```

where:

FNletter [$]
Is the name of the defined function which must consist of FN followed by a letter from A to Z. An optional dollar sign denotes a function with a string result.

(param-list)
variable [,variable. . .]

local-list
variable [,variable. . .]

Programming Notes:

1. Any reference to a defined function for which the user has not supplied a corresponding DEF statement is treated as an error.

2. The redefinition of a defined function is treated as an error.

3. A defined function may reference any other function except itself. Recursive definitions are not allowed.

4. A function may be invoked only from an expression.

5. The param-list is used to pass values in one direction only and that is to the function. Variables in the param-list are local. Variables in the param-list may be string or numeric in type. When called, the passed parameters in the call and in the definition must have matching types.

6. If the function definition requires several statements (multiline function), the DEF statement defines the entry into the function and requires a unique, corresponding FNEND statement that defines the exit from the function. Branching into and out of a multiline function definition or branching to a DEF statement is illegal.

7. A local-list can be provided for a multiline function to indicate that the variables named in the list are to be local only throughout the function definition. Such variables may be used for any other purpose outside the function definition; upon entry into the function, the variables are initialized to zero.

8. To give a multiline function a value, the function name must appear to the left of an equal sign in an assignment statement.

9. A DEF statement within a function definition is illegal.

10. The param-list and local-list variables are restored to their original values upon exiting from the function definition.

11. All function definitions containing local parameters must appear before they are referenced by the main program. If a DEF statement is encountered during normal program flow, the statements defining the function are bypassed and control passes to the next statement within the main program.

12. If no parameters are to be passed to the function, the param-list may be omitted.

13. The function may reference variables external to it by using the same variable name as was used in the main program.

14. Functions that are passed in subprogram CALLs must be defined prior to the CALL statement.

Example 1:

```
30 DEF FNE(X) = EXP(-X**2)
```

During execution, this statement would be invoked for various values of the function e^{-x^2} by referencing FNE(.1), FNE(3.45), FNE(A + 2), etc. Such a definition can simplify the program when values of some function are needed for a number of different values of the variable.

Example 2:

```
100 DEF FNA$,B$
110 PRINT ''ENTER YES OR NO'';
120 INPUT B$
130 IF B$=''NO'' THEN 150
140 IF B$< > ''YES' THEN 110
150 FNA$=B$
160 FNEND
.
.
.
1650 IF FNA$ = ''YES'' GOTO 2000
```

This multilined string function allows the user to request, accept, and answer by referencing the user function FNA$.

DIM

3.2.2. DIM Statement

The DIM statement explicitly specifies the upper bounds of numeric and string arrays to reserve sufficient space in main storage for the array. Either a 1- or 2-dimensional numeric array or a 1-dimensional string array can be dimensioned. The lower bound for each dimension is always 0.

Format:

```
DIM {numeric-dimension} [,{numeric-dimension}...]
    {string-dimension }    {string-dimension }
```

where:

`numeric-dimension`
Is a letter followed by one to five digits in parentheses or a letter followed by two numbers (each consisting of one to five digits) separated by a comma in parentheses.

`string-dimension`
Is a letter followed by a dollar sign ($) followed by one to five digits in parentheses; or a letter followed by a dollar sign, followed by two numbers (each consisting of one to five digits) separated by a comma in parentheses.

Programming Notes:

1. The duplication of an array name in a DIM statement is treated as an error.
2. The appearance of the same array name in more than one DIM statement is treated as an error.
3. If the value of a subscript of an array exceeds 10, the array name must appear in a DIM statement; otherwise, an error occurs.
4. A DIM statement can appear anywhere in the program, and may appear after the related variable is used, providing the number of subscripts remains consistent.
5. The upper limit on the subscripts of an array is referred to as the array dimensions or dimensions of the array.
6. Numeric array elements are initialized to zero and string elements to null strings.
7. The DIM statement defines the maximum bounds for the array. Certain other statements may be used to change the array bounds dynamically during execution. Changing the array bounds will limit the set of elements that can be referenced by subscripts or matrix operations.

Example 1:

```
20 DIM A(25)
```

In this example, A is a 1-dimensional numeric array consisting of 26 numeric variables: A(0), A(1), . . .,A(25).

Example 2:

```
21 DIM B(20,30), R$(35)
```

In this example, B is a 2-dimensional numeric array consisting of 651 numeric variables:

```
B(0,0),B(1,0),...,B(20,0)
B(0,1),B(1,1),...,B(20,1)
   .
   .
   .
B(0,30),B(1,30),...,B(20,30)
```

and R$ is a 1-dimensional string array consisting of 36 string variables: R$(Ø), R$(1),...,R$(35).

FNEND

3.2.3. FNEND Statement

The FNEND statement terminates a multiline function and is the only way of exiting from a multiline function. All variables in the local-list and param-list in the DEF statement are restored to their values before the function call.

Format:

```
FNEND
```

Programming Notes:

1. Each multiline function must terminate with exactly one FNEND statement.

2. Multiple FNEND statements for a given DEF statement are illegal.

Example:

```
25 DEF FNE (A,B,C),D
30 D=A*5
35 FNE=A + B + C + D
40 FNEND
```

This example illustrates a multiline function. A, B, and C are the param-list variables, while D is the local-list variable of the multiline function FNE. As shown, the multiline function must begin with a DEF statement and terminate with an FNEND statement.

3.3. REMARK STATEMENT (REM)

The REM statement provides a means for inserting explanatory remarks into a program. Although what follows REM is ignored, its line number may be used in a control statement. Comments may also be appended to BASIC statements by prefixing the comment with an apostrophe.

Format:

```
REM [character . . .]
```

Example:

```
100 REM INSERT DATA IN LINES 900-998. THE FIRST
110 REM NUMBER IS N, THE NUMBER OF POINTS. THEN
120 REM THE DATA POINTS THEMSELVES ARE ENTERED, BY
.
.
.
200 REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS
```

LET

3.4. ASSIGNMENT STATEMENT (LET)

The LET statement assigns a value to a variable.

Format:

```
[LET] { numeric-let   }
      { string-let    }
      { function-let  }
```

where:

numeric-let
Numeric-reference = [numeric-reference=...] arithmetic-expression.

string-let
String-reference = [string-reference=...] string-expression.

function-let
FN letter [$] = expression.

Programming Notes:

1. The statement verb LET need not be written.
2. Mixed mode assignment is not accepted by the syntax checker.
3. Multiple assignments are allowed. The right-hand expression is evaluated and then assigned to each of the references, from right to left, in turn. Subscripts are evaluated just prior to any assignments within the current statement.
4. The function-let assigns a value to a multiline user-defined function. (See DEF statement.)

Example 1:

```
10 LET I=2
20 A(I)=I=3.5
```

Statement 20 assigns I the value 3.5 and then A(I) is assigned the value of I, which is 3.5.

Example 2:

```
56 LET G$=H$=''THIS STRING''
```

This is a string-let statement that assigns the closed string constant "THIS STRING" to string variable H$, which in turn is assigned to string variable G$.

Example 3:

```
10 DEF FNA(A,B,C,D)
20 LET FNA=(A-B)*(C+D)
30 FNEND
```

Statement 20 is a function-let statement used in the multiline function FNA.

3.5. CONTROL STATEMENTS

These statements give the programmer the ability to alter and control the normal sequence of statement execution. Included in this group of statements are: END, FOR and NEXT, GOSUB and RETURN, GOTO, IF, ON, PAUSE, STOP, RANDOMIZE, TIME, and SYSTEM statements.

END

3.5.1. END Statement

The END statement is the last statement in a BASIC program.

Format:

```
END
```

Programming Notes:

1. When the user issues the RUN command, all statements up to and including the END statement, and any subprograms which may follow, are compiled.

2. Only one END statement may be present in a program. Any statements after the END are treated as an error.

Example:

```
*30 END
```

FOR and NEXT

3.5.2. FOR and NEXT Statements

The FOR statement initiates a loop; the NEXT statement, whose variable matches the one specified in the FOR statement, terminates the loop.

Format:

```
FOR numeric-variable=arithmetic-expression TO arithmetic-expression
[STEP arithmetic-expression]
NEXT numeric-variable
```

Programming Notes:

1. A FOR-NEXT loop specifies the iteration of a sequence of statements for given values of the numeric-variable (loop index). The initial, final, and step values are given by the three arithmetic expressions specified in the FOR statement. A step value of +1 is assumed if the STEP is omitted. These values are calculated on each entry into the loop.

   The loop index may be used in calculations within a FOR-NEXT loop. In particular, its value may be changed by assignment and this will affect the sequence of values for which the loop is iterated.

   Let i, f, s, c designate the initial, final, step, and current values, respectively, of a loop index.

   Then, initially, we must have $(f-1)*s \geq 0$. That is, the step value, which may be negative, must move the loop index value in the direction of the final value.

   If $f > i$ and $s = 0$, then program execution will continue indefinitely within the FOR-NEXT loop. The calculations to determine loop termination are done at the top of the loop; thus, the statements in the FOR-NEXT loop may be skipped entirely.

   If control is transferred into a FOR-NEXT loop, the results are unpredictable.

2. In the NEXT statement, the numeric-variable must be the same as that following the verb FOR in the FOR statement. If a different numeric-variable is detected (indicating an overlapping nested loop), an error results. An error will also result because of any one of the following conditions:

   a. The occurrence of a NEXT statement prior to its corresponding FOR statement.

   b. A FOR statement without its corresponding NEXT statement.

   c. More than one FOR statement with the same index (variable) prior to the occurrence of the NEXT statement corresponding to the first such FOR statement (that is, loops may be nested, but not if they use the same index).

3. The TO and STEP operand order is not checked.

   Example:

```
10 FORI=1 TO 10 STEP 2
```

   is the same as

```
10 FORI = 1 STEP 2 TO 10
```

4. Nesting is allowed to 10 levels.

   Example:

```
 30 FOR X = 0 TO 3 STEP D
 80 NEXT X
120 FOR X4=(17+COS(Z))/3 TO 3*SQR(910) STEP 1/4
235 NEXT X4
240 FOR X = 8 TO 3 STEP -1
 .
 .
 .
300 NEXT X
456 FOR J = -3 TO 12 STEP 2
 .
 .
 .
500 NEXT J
```

   Note that the step size may be a fraction (1/4), a negative number (-1), or a positive number (2). In the example with lines 120 and 235, the successive values of X4 will be .25 apart, in increasing order. In the next example (lines 240 through 300), the successive values of X will be 8, 7, 6, 5, 4, 3. In the last example (lines 456 through 500), J will take on values -3, -1, 1, 3, 5, 7, 9, and 11.

5. The action of the FOR statement and the NEXT statement is defined in terms of other statements as follows:

   FOR v = initial-value TO limit STEP increment

   (block)

   NEXT v

is interpreted as

```
                    LET own1 = limit
                    LET own2 = increment
                    LET v = initial-value
line 1                  IF(v-own1)*SGN(own2)>0 THEN line 2

                    (block)

                    LET v = v + own2
                    GOTO line 1
line 2                  (continue in sequence)
```

GOSUB and RETURN

3.5.3. GOSUB and RETURN Statements

The GOSUB statement provides a subroutine call facility.

Format:

```
GOSUB line-number
RETURN
```

Programming Notes:

1. The GOSUB statement transfers control to the statement whose line number is referenced. Control is subsequently returned to the statement following the GOSUB by executing a RETURN statement.

2. A GOSUB statement inside a subroutine may be used to call another routine. This is referred to as *nested GOSUBs*. It is necessary that the RETURN statement be used to exit from the subroutine. The execution of a RETURN statement before a GOSUB statement is treated as an error.

3. GOSUB and RETURN statements need not be paired; that is, the same RETURN statement may be used to return from several different GOSUBs.

Example:

```
90 GOSUB 210
91 A=3.1
    .
    .
    .
100 STOP
210
     .
     .
     .
350 RETURN
```

The GOSUB statement (line number 90) directs the system to line number 210, which is the first statement of a subroutine. The last statement of the subroutine is line number 350 (a RETURN statement), which causes the system to return to line number 91 of the program.

3.5.4. GOTO Statement

The GOTO statement unconditionally transfers control to the statement whose line number is referenced.

Format:

```
GOTO line-number
```

Programming Note:

The nonexistence of the statement whose line number is referenced is treated as an error.

Example:

```
19 LET J$=''THIS STRING''
20 GOTO 25
21 READ A$,B$,C$
        .
        .
        .
25 K$=''WHAT STRING''
```

The GOTO statement (line number 20) transfers control to the assignment statement (line number 25) and bypasses the READ statement (line number 21).

IF

3.5.5. IF Statement

The IF statement conditionally transfers control. When the condition specified is true, control is transferred to the line number referenced.

Format:

```
IF condition {GOTO } line-number
             {THEN }
             {GOSUB}
```

where:

condition
Is one of the following:

arithmetic-expression relation arithmetic-expression
string-expression relation string-expression
END channel-setter
MORE channel-setter

relation
Is any of the symbols listed in Table 3-2.

Table 3—2. Relation Symbols

Symbol	Meaning	Example
=	Is equal to	A = B
<	Is less than	A < B
< = = <	Is less than or equal to	A < = B A = < B
>	Is greater than	A > B
= > > =	Is greater than or equal to	A = > B A > = B
< > ≠	Is not equal to	A < > B A ≠ B

Programming Notes:

1. Mixed mode expressions across a relation are not accepted by the syntax checker.

2. When two strings of different lengths are compared, the shorter string will be padded on the right with blanks until it is of equal length to the longer string. Thus, string comparison is always performed on equal length strings. This results in correct collating sequence. Note that this logic of string comparisons does not affect the actual stored lengths or values of strings. Also, null strings are considered to be a string of all blanks in all string comparisons.

3. The condition may test two arithmetic or two string expressions against each other using the tests listed in Table 3-2. If the condition is met, the transfer is completed.

4. The condition may also be a file test, in which case the specified file is tested to see if there are MORE records left to be read, or if the file is at END. The channel setter specified must refer to an open file. If the file has not been opened by a file statement, execution will be terminated.

5. If the last record of a file has been read, but not entirely processed, the IF END statement will test true. That is, the file is considered to be at end of file if no additional READ is permitted. However, there may still be data in the buffer that an INPUT would accept.

Example 1:

```
10 A$=''ASHLEY''
20 B$=''BOB''
30 IF A$<B$ THEN 50
40 STOP
   .
   .
   .
50 PRINT A$;B$
   END
```

In this example, string A$ is smaller in value than string B$, although string A$ is greater in length than string B$. Thus, control transfers to line number 50 after executing the IF statement on line number 30.

Example 2:

```
40 IF SIN (X)=M THEN 80
```

In this example, if the sine of X is equal to M, control transfers to the statement with line number 80.

ON

3.5.6. ON Statement

The ON statement provides a multibranched switch.

Format:

```
ON arithmetic-expression {GOTO  } line-number[,line-number...]
                         {GOSUB }
                         {THEN  }
```

Programming Notes:

1. The arithmetic expression is rounded to the nearest integer; it is used as the index to select and branch to one of the sequence of line numbers.

2. If the value or the arithmetic expression is less than 1 or greater than the number of line numbers specified, a run-time error results.

3. Once the selection has been determined, a GOTO, a THEN, or a GOSUB is performed. In the case of a GOSUB, a RETURN will return to the next statement.

Example:

```
150 ON X+Y GOTO 575,490,650
2170 ON FNA(G) GOSUB 2200,2400
```

The first statement transfers control to line number 575, 490, or 650, depending upon whether the integer part of the expression X+Y yields 1, 2, or 3, respectively.

The second statement will execute either a GOSUB 2200 or a GOSUB 2400, depending on whether FNA (G) has a value of 1 or 2.

3.5.7. PAUSE Statement

The PAUSE statement interrupts program execution and causes the following message to be typed out at the terminal:

```
PAUSED AT line-number CONTINUE(Y OR N)?
```

If the user responds with N or NO, execution is terminated. If the user responds with Y or YES, execution is to be continued at the next sequential line number.

Format:

```
PAUSE
```

Example:

```
*10 PRINT  ''THIS IS A TEST PROGRAM''
*20 PAUSE
*30 PRINT  ''THIS IS ANOTHER LINE''
*40 PAUSE
*50 END
*   RUN
    THIS IS A TEST PROGRAM
    BA063 EXECUTION PAUSED AT LINE 00020 CONTINUE (Y,N)? ▷  Y
    THIS IS ANOTHER LINE
    BA063 EXECUTION PAUSED AT LINE 00040 CONTINUE (Y,N)? ▷  N
*
```

STOP

3.5.8. STOP Statement

The STOP statement is used to halt program execution and causes the following message to be typed out at the terminal:

```
EXECUTION STOPPED AT line-number
```

Format:

```
STOP
```

Programming Note:

A STOP statement may appear anywhere in the program.

Example:

```
*10 INPUT A
*20 IF A = 10 THEN 40
*30 STOP
*40 PRINT ''KEEP GOING''
*50 END
*   RUN
?12
    BA062 EXECUTION STOPPED AT LINE 00030
```

3.5.9. RANDOMIZE Statement

This statement will generate a random seed for use by the random number generator. Its function is equivalent to the function call RND(-1). If not used, a given sequence of calls to RND will generate the same sequence of numbers for repeated executions.

Format:

```
RANDOMIZE
```

Example:

```
10 RANDOMIZE
```

TIME

3.5.10. TIME Statement

This is a nonexecutable statement specifying the maximum CPU seconds allowed for program execution. If multiple TIME statements occur, the minimum value specified is used. When the specified time limit is reached, the following message is displayed:

```
BA059 TIME UP --- PROGRAM LOOPING
```

Format:

```
TIME integer
```

where:

integer
Specifies an integer number of CPU seconds.

Example:

```
5 TIME 150
```

3.5.11. SYSTEM Statement

This is an executable statement that allows a BASIC program to issue any system command.

Format:

```
SYSTEM ''system commandΔ''
```

Programming Notes:

1. The contents of the closed string should not start with a slash and should end with at least one space.

2. Errors that occur will be displayed on the user's terminal, but will not be reported to the BASIC program.

Example:

```
193 SYSTEM ''FSTATUS $Y$SRC, REL070 ''
```

3.6. DATA INPUT/OUTPUT STATEMENTS

The input/output statements permit the user to transfer data between internal storage and the terminal, print data at the terminal (and format the data), and use the same data in a program as many times as required. The input/output statements are: INPUT, LINPUT, MARGIN, PRINT, READ and DATA, RESTORE and RESET, and USING.

The following subsections present these statements in their simplest form for use with terminal input/output and program supplied data. These and additional statements are presented in Section 4.

INPUT

3.6.1. INPUT Statement

Data may be entered dynamically during the running of a BASIC program using the INPUT statement.

Format:

```
INPUT variable [,variable . . .]
```

where:

`variable`
Is either a numeric or string variable reference. This may be either a scalar variable or a reference to an array element.

Programming Notes:

1. The INPUT statement is similar to the READ statement, except that its data is input (dynamically) from the user's terminal. The user is prompted for input data by a question mark (?). Insufficient data results in additional prompting. Data must be entered according to the type of variable in the INPUT statement. Data items entered must be separated by commas. The inputting of invalid data causes an error message to be printed at the user's terminal. In this case, the user must retype the data starting with the data item in error.

2. If the first four characters of the input are STOP, program execution is terminated.

Example:

```
20   PRINT ''TYPE IN VALUES FOR X, Y, AND Z'';
30   INPUT X, Y, Z
```

Execution of these statements causes the system to type out the following message:

```
TYPE IN VALUES FOR X, Y, AND Z?
```

The terminal device would be positioned after the question mark waiting for input values for X, Y, and Z. Note that without the semicolon at the end of line number 20, the question mark would have been posted on the next line.

3.6.2. LINPUT Statement

The LINPUT statement allows an entire input line to be read into a single string variable. No input checking or conversion is performed.

Format:

```
LINPUT string-variable [,string-variable . . .]
```

where:

`string-variable`
Is a reference to a simple string variable or a string array element.

Example:

```
10      LINPUT    C$,H$(6,5)
```

This statement will cause the user to be prompted twice for input. The first input response will be stored in its entirety in variable C$. The second response will be stored in array element H$(6,5).

MARGIN

3.6.3. MARGIN Statement

The MARGIN statement sets the current margin for the terminal.

Format:

```
MARGIN      numeric-expression
```

Programming Notes:

1. The value of the numeric expression in the MARGIN statement is truncated, and the resulting integer is used for the output margin length for the terminal.

2. The MARGIN statement takes effect immediately, even if a line of output is partially filled.

3. The width of a terminal line defaults to 80 characters unless reset by a MARGIN statement.

Example:

```
1       MARGIN   64
```

This statement sets the current margin to 64 characters. This may be useful for UNISCOPE terminals with 64 character lines.

3.6.4. PRINT Statement

The PRINT statement results in data items being printed at the user's terminal.

Format:

```
PRINT, item[{,} item . . .] [{,}]
           [{;}             ] [{;}]
```

where:

item
: Is an expression or TAB (expression).

Programming Notes:

1. The width of a printed line on a user's terminal defaults to 80 characters, but may be reset by a MARGIN statement.

2. Using the comma (,) or the semicolon (;), it is possible to control horizontal positioning on a printed line. Initially, the print line is divided into fields of 15-character positions each.

   a. If a comma is used after an item, the next item will be printed in the next available field. A data item is placed at the beginning of a field. If an item cannot be placed in a field because it will cause the line to exceed the maximum print positions for a device, then that item will be placed in the first field on the next line. If the last item in the current PRINT statement is followed by a comma or semicolon, and there is sufficient space remaining on the line, then the items in the next PRINT statement will be printed on the same line. If the last item is not followed by a comma or semicolon, then the next PRINT statement begins printing on a new line.

   b. If a semicolon is used after an item, the next item will be printed in the next print position on the line (i.e., the item following the string is printed directly connected to it).

   c. For numeric items, the size of a zone depends upon the number of digits needed to represent the data item. The zone width is always one character more than is needed for the data item. In each case, the number is printed starting at the first position of the zone. Numbers that cannot be represented as six or fewer digits are represented in E-notation (refer to Programming Note 5) and occupy either 11 or 12 print positions within a 13-position zone.

3. Whenever the TAB function is used in the PRINT statement, it will cause the print head to move over to the position indicated by the integer value of the TAB expression. The use of the comma and the semicolon remains unchanged in this type of statement. When a comma follows a variable, a fixed field width is reserved before the next entry in the statement is recognized. The semicolon causes this field width to be minimized. Thus, when the terminal device is being tabbed, the semicolon should be used. The TAB expression is evaluated modulo the current margin size; a value less than or equal to zero results in an error. If the value of the TAB expression is less than the current print position, the current line is printed and a new line is begun.

4. When a string reference is encountered that has not been assigned (a null string), the PRINT statement will produce no printout.

5. The conventions for printing numeric data are as follows:

   a. An integer number is printed as an integer.

   b. In all cases, no more than six significant digits will be printed.

   c. If the number is positive, the sign is not printed, but a print position is left blank.

   d. Decimal numbers will be printed without an exponent part whenever possible. Decimal numbers requiring an exponent field will be printed:

      -#.#[###]E ± dd

      where the mantissa may be up to six digits. Trailing zeros in the mantissa are not printed.

   e. A space follows every number printed.

6. If no items are present on the PRINT statement, a line advance occurs.

Example 1:

```
10   FOR X = 1 TO 15
20   PRINT X
30   NEXT X
40   END
```

This example prints the numbers 1 to 15 on 15 lines as follows:

```
Col 1
↓
△1
△2
△3
△4
△5
△6
△7
△8
△9
△10
△11
△12
△13
△14
△15
```

Example 2:

```
10    FOR X = 1 TO 15
20    PRINT X,
30    NEXT X
40    END
```

This example prints the numbers 1 to 15 in 3 lines as follows:

```
Col 1      Col 16     Col 31     Col 46     Col 61
  ↓          ↓          ↓          ↓          ↓
△1         △2         △3         △4         △5
△6         △7         △8         △9         △10
△11        △12        △13        △14        △15
```

Example 3:

```
10    FOR X = 1 TO 15
20    PRINT X;
30    NEXT X
40    END
```

This example produces a single line of printout of the numbers 1 to 15 as follows:

```
△1△△2△△3△△4△△5△△6△△7△△8△△9△△10△△11△△12△△13△△14△△15
```

If statement 20 were modified, the following would be printed:

```
20 PRINT - X;
-1△-2△-3△-4△-5△-6△-7△-8△-9△-10△-11△-12△-13△-14△-15
```

Example 4:

```
20    LET A = 1
30    C$ = ''SALESMAN''
40    A$ = ''JOE''
50    B$ = ''△DOKES''
60    N = 4
70    PRINT A, -16,A$;B$,C$;N
80    END
```

The execution of statement number 70 would produce the following output line:

```
Col.  Col   Col Col.        Col.       Col.
 2     16    31  35         46         55
 ↓     ↓     ↓   ↓          ↓          ↓
 1    -16   JOE△DOKES      SALESMAN△4
```

Example 5:

```
10   PRINT ''0000000001111111111222222222233333333333''
20   PRINT ''12345678901234567890123456789012345678901''
30   A$=''*''
40   A = 1
50   PRINT TAB (9);A
60   PRINT TAB (19);A
70   PRINT TAB (29);A
80   PRINT TAB (10);A$;TAB(20);A$;TAB(30);A$
90   END
```

This example illustrates the use of the TAB function in the PRINT statement. The output of this program is as follows:

```
Col 1     Col 10     Col 20     Col 30
↓         ↓          ↓          ↓
0000000001111111111222222222233333333333
12345678901234567890123456789012345678901
          1
                     1
                                1
          *          *          *
```

Example 6:

```
10   FOR X = 1 TO 25
20   PRINT 2**X;
30   NEXT X
40   END
```

This is an example of how large numbers are printed and how they are spaced when a semicolon is used in the PRINT statement. The printout produced is as follows:

```
△2△△4△△8△△16△△32△△64△△128△△256△△512△△1024△△2048△△4096△△8192△△16384△△32768
△65536△△131072△△262144△△524288△△1.04858E+06△△2.09715E+06△△4.1943E+06
△8.38861E+06△△1.67772E+07△△3.5544E+07
```

READ and DATA

3.6.5. READ and DATA Statements

The READ statement assigns values to the listed variables. These values are obtained from the DATA statement.

Format:

```
READ {string-variable }[,{string-variable }...]
     {numeric-variable}[ {numeric-variable} ]
DATA {string-constant }[,{string-constant }...]
     {numeric-constant}[ {numeric-constant} ]
```

Programming Notes:

1. Before the program is run, BASIC takes all of the DATA statements in the order in which they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available datum (or data). The string data block supplies values for string variables, and the numeric data block supplies values for numeric variables.

2. Insufficient data results in program termination with a diagnostic message.

Example:

```
10   READ X,Y,Z,X1,Y2,Q9
20   DATA 4,2,1.7
30   DATA 6.734E-3,-174.321,3.14159265
35   PRINT X,Y,Z,X1,Y2,Q9
40   FOR K=1 TO 5
50   READ B
55   PRINT B
60   NEXT K
71   DATA 1
72   DATA 2
73   DATA 4
74   DATA 5
75   DATA 1.234E16
80   END
```

The execution of the above example would produce the following output:

```
Col          Col          Col          Col          Col
2            16           31           46           61
↓            ↓            ↓            ↓            ↓
4            △2           △1.7         △.ØØ6734     -174.321
3.14159
1
2
4
5
1.234E+16
```

3.6.6. RESTORE and RESET Statements

The RESTORE and RESET statements permit the user to read data from the beginning of a data block.

Format:

```
RESTORE
RESET
```

Example:

```
 10    READ N
 20    FOR X = 1 TO N
 30    READ X
       .
       .
       .
100    NEXT X
110    RESTORE
120    READ M
130    FOR J = 1 TO M
140    READ Y
       .
       .
       .
200    NEXT J
300    DATA 5
310    DATA 1.0
315    DATA -01
320    DATA 3.2E+01
325    DATA 4
330    DATA -3.
400    END
```

In this example, the READ statements on line numbers 10 and 120 will read the same datum (i.e., the number 5 contained in the DATA statement on line number 300). Similarly, the READ statements on line numbers 30 and 140 will read the same data from the DATA statements on line numbers 310 to 330.

USING

3.6.7. USING Statement

The PRINT USING format of the PRINT statement gives the BASIC user the ability to define the format of his program's output. The USING clause consists of three parts: the USING keyword, the using string that contains the format fields, and the expression-list that is used to fill in the format fields of the using string.

Format:

```
USING using-string,expr-1,expr-2,...,expr-n
```

Example:

```
PRINT USING ''<####=STRING FIELD,+##=NUMERIC FIELD'',S1$,N
```

As shown, both string and numeric output can be formatted by a using string. Numeric fields begin with a $, +, or –, and can only contain numeric output. String fields begin with < or >, and only string data can be formatted into a string field. Each starting character has a defined function and will be explained later. The # is a place holder and, by varying the number of place holders, the user can change the size of the format field and thus the format of the output.

A format field begins with one of the characters $, +, –, <, or > and contains all characters up to but not including the next $, +, –, <, or > (or to the end of the using string). The complete using string may be made up of numerous format fields. A format field can appear anywhere within a using string and the place holders do not have to be contiguous. If more format fields are given in the using string than variables in the variable-list, the excess fields are ignored. If there are extra variables in the list, then the using string will be reused until the variable-list is exhausted.

Any characters that do not have special meanings as described in this section may be embedded within format fields. As the BASIC system edits data into the place holders, any embedded characters are copied too.

Example 1:

If variable S$ contains the string:

```
''A=+##,B=-##,AND C$ CAN=<### OR ###''
```

the statement

```
PRINT USING S$,20,-20,''ABCDXYZ''
```

would produce the following output:

```
A=+20,B=-20,AND C$ CAN=ABCD OR XYZ
```

Example 2:

If only one variable is printed, the result would be:

```
PRINT USING S$,20
A=+20,B=
```

Example 3:

```
PRINT USING S$,-20,20,''ABCDXYZ'',30,-30
```

will output:

```
A=-20,B=20,AND C$ CAN = ABCD OR XYZA=30,B=-30,AND C$ CAN =
```

3.6.7.1. Formatting String Output

The BASIC user has two options for formatting the string output of the BASIC program. He can left-justify or right-justify the output in the format field defined in the using string.

The format field must start with a < to left-justify the output. When a format field starts with this character, the field is filled from left to right starting with the leftmost character, in this case the <, until the format field or the string is exhausted. If the string is not long enough to fill all of the place holders, then the remaining place holders are space-filled. If there are more characters in the string than there are place holders, the string is truncated.

If the format field starts with a >, then the string is right-justified in the format field. The last place holder in the field is replaced with the last character of the string being printed. The next to the last place holder is filled with the next to the last character and so on from right to left until the format field is completely replaced by the string. If the format field is longer than the string being printed, the remaining place holders, including the > are replaced by spaces. If the string is longer than the format field, the leftmost characters of the string are omitted.

Example 1:

```
PRINT USING ''|<######|'',''ABCD''
```

will output:

```
|ABCD      |
```

Example 2:

```
PRINT USING ''|>######|'',''ABCD''
```

will output:

```
|      ABCD|
```

3.6.7.2. Formatting Numeric Output

Through using strings, the BASIC user is given a wide variety of ways to format numeric output. The user can dictate the number of decimal places that are printed, thus defining the accuracy of the number being output. An exponent field can be defined in order to neatly print large numbers. The numeric field can be preceded by three different field descriptors. A dollar sign causes the dollar sign to be right-justified against the output number. The plus sign right-justifies a plus sign against the number if the number is positive, or a minus sign if the number is negative. A minus sign causes a minus sign to be right-justified if the number is negative; if the number is positive, no sign is printed. To further identify the output, the user can combine the dollar sign with a plus or minus sign, giving $+ or $-. Examples will be given later to explicitly show each format that can be used.

Many different situations can occur when printing numbers with format fields due to the flexibility in describing the format fields and the varying magnitude of the numbers being printed. The following paragraphs present some of these situations and explain how each is handled.

When a numeric field is defined, the user should be aware of the expected magnitude of the number to be printed in the field. The magnitude of a number cannot be greater than the size of the format field (number of place holders) in which the number is to be printed. An example would be printing the number 100 in the format field +##. In this field there are only two numeric positions, and the 100 will take three. To inform the user that this error has occurred, the entire format field is replaced by asterisks. In this case, the output is ****.

There are two ways to avoid this problem. First, the format field can be made very large in order to accommodate large numbers. This is an adequate solution, but can lead to another problem. BASIC will only print six significant figures; if the user attempts to print more than six significant figures (an example would be 10000000), then the number is truncated to six figures and the remaining portion of the format field is replaced with question marks. Output printed in this manner may not always be in good readable form. In the example previously given, if the format field used was +##########, the output is +100000???.

A second method for printing numbers of varying magnitudes avoids using large format fields by defining an exponent field in the format string. An exponent field is defined by five consecutive up-arrows ↑↑↑↑↑. When an exponent field is used, the number is adjusted to fit into the defined field, and the exponent is then calculated to give the user the magnitude of the number. If an exponent field is defined in the format string, such as +###↑↑↑↑↑, then the magnitude of the number is known. The +1000000 is formatted as +1000 E+03 and the +100000000 is printed as +1000 E+05 which tells the user exactly what was printed. As seen in the examples, the exponent field in the format field is formatted as follows:

```
space E sign digit digit
```

If an exponent is used with a numeric format field, then any number can be printed in the field. The number is adjusted to the field size, and the exponent holds the magnitude of the adjusted number. If this statement is executed:

```
157 PRINT USING''+##↑↑↑↑↑'',25,290,-300,.00001
```

the result is:

```
 +25 E+00 +29 E+01 -30 E+01 +10 E-06
```

To print numbers that contain a decimal component, the user can define decimal fields in the format field. The format field begins with a +, -, $, $+, or $-, optionally followed by any number of place holders. A decimal point may be embedded anywhere within the place holders. The following field will contain a decimal field of three places, "+##.###". When the decimal is printed, it is rounded to the number of positions given and then printed. When no decimal places are given, the number is rounded to the next integer value.

Examples:

Format Field	Number Printed	Resulting Output
+#####	+100	+100
+#####	-100	-100
-#####	+100	100
-#####	-100	-100
$###.##	+20.99	$20.99
$+###.##	-20.99	$-20.99
$+###.##	+20.99	$+20.99
$-###.##	+20.99	$20.99
$### AND ## CENTS	+45.50	$45 AND 50 CENTS
DICE - AND -	1,1	DICE 1 AND 1
$#,###.##	1234.56	$1,234.56
$#,###.##	8.94	$8.94
-#:00 HOURS ## MINUTES	1234	12:00 HOURS 34 MINUTES
TODAY IS THE -#TH OF SEPT, 19##	2680	TODAY IS THE 26TH OF SEPT, 1980

3.6.7.3. Use with PRINT Statement

The USING clause may only be used in combination with a PRINT or MAT PRINT statement. As previously stated, a USING clause begins with the word *USING*, followed by a string and a list of expressions to be formatted:

```
USING string-expression, expression, expression, ...
```

Examples:

```
106 PRINT USING A$,B,C,10,E(5)
107 PRINT USING ''FILES-#DISKS-#TAPES-#'',F,D,T
108 PRINT USING ''USER RESPONSE OF>#### IS INVALID''.U$
109 PRINT USING FNB$(6),T,U,SIN(3.14159)
```

The USING clause need not be the only thing on a PRINT statement; unformatted expressions may be combined with formatted data. When combining formats in this manner, it is important for the user to realize exactly where a USING clause begins and ends. It always begins with the word USING. The end of the USING clause occurs either at the end of the PRINT statement that contains no trailing comma or at a semicolon.

When a USING clause is encountered, BASIC formats the entire using string and the PRINT statement prints it to the output device. Thus, when used with files, the using string, after editing, must not be longer than the margin for the file.

Examples of combined formats are shown; the shaded areas indicate the USING clauses.

```
243 PRINT #1: USING A$,B,C;D,E$
244 PRINT USING A$,B,C;D,E$
246 PRINT TAN(X), USING ''IS THE TANGENT OF -###.#!!!!!'',X
247 LET F$ = ''IS THE <######## OF -###.#!!!!!''
248 PRINT TAN(X);USING F$, ''TANGENT'',X;SIN(X);USING F$,''SINE'',X
```

The list of expressions to be used with a single USING clause can be extended over several PRINT statements by ending the statements with a comma. This indicates that more expressions are to follow, and BASIC will delay printing the output until a semicolon is found in a subsequent PRINT, or until a PRINT is executed that does not end with a comma.

Examples:

```
341 PRINT USING A$,B,C,D,
342 PRINT E,F;G,H
343 PRINT USING I$,J$,K,L(3),
344 PRINT SIN(3.14159),
345 PRINT M
346 PRINT N,O
347 PRINT P,USING Q$,R;
348 PRINT S
```

Variables B, C, D, E, and F are printed under the format in A$, variables G and H are unformatted. Variables J$, K, array element L(3), the sine of 3.14159 and variable M are under the format in I$, while N and O are unformatted. Variables P and S are unformatted, while R is printed under the format in Q$.

The final example of the USING clause shows how the format fields are reused when insufficient format fields exist for all of the variables to be printed.

Examples:

```
179 PRINT USING''-.###!!!!! IS THE <###### OF-.###!!!!!'',TAN(X),''TANGENT'',
180 PRINT X,SIN(X),''SINE'',X,COS(X),''COSINE'',X
181 PRINT COS(X);''IS THE COSINE OF'';X
4.855E+05 IS THE TANGENT OF 1.571E+00 1.000E+00 IS THE SINE OF 1.571E+00
2.060E-06 IS THE COSINE OF 1.571E+00
2.05959E-06 IS THE COSINE OF 1.57079
```

Because statement 179 ends with a comma, the USING clause is still active. Any variables printed on a succeeding PRINT statement will still be under format control. Statement 180 does not end with a comma, so it terminates the format. A total of nine expressions is formatted. Statement 181 is a normal PRINT statement.

This example shows several unique properties of USING clauses. The format string contains three format fields:

```
-.###↑↑↑↑ IS THE
<###### OF
-.###↑↑↑↑
```

MAT

3.7. MATRIX OPERATION STATEMENTS

For ease in handling matrix operations on numeric arrays, the following MAT statements are provided in BASIC:

- MAT addition, subtraction, and multiplication statements

  `MAT C = A + B`
  Add the two matrixes A and B; store the result in matrix C.

  `MAT C = A - B`
  Subtract matrix B from matrix A; store the result in matrix C.

  `MAT C = A * B`
  Multiply matrix A by matrix B; store the result in matrix C.

- MAT constant statement

  `MAT C = CON`
  Set each element of matrix C to a value of 1.

- MAT identity statement

  `MAT C = IDN`
  Set the diagonal elements of matrix C to 1's, and all other elements to 0, yielding an identity matrix.

- MAT INPUT statement

  `MAT INPUT A,A$`
  Input elements of a matrix.

- MAT inversion statement

  `MAT C = INV (A)`
  Invert matrix A; store the resulting matrix in C.

- MAT LINPUT statement

  `MAT LINPUT A$,B$`
  Input lines of data into elements of matrixes using the LINPUT statement.

- MAT null statement

  ```
  MAT C$ = NUL$
  ```
  Set each element in matrix C$ to a null string.

- MAT PRINT statement

  ```
  MAT PRINT A,A$
  ```
  Print elements of matrix A.

- MAT READ statement

  ```
  MAT READ A,A$
  ```
  Read elements of matrix A from DATA statements.

- MAT scalar multiply statement

  ```
  MAT C = (exp)*A
  ```
  Multiply each element of matrix A by the value of the expression and place the result in matrix C.

- MAT transpose statement

  ```
  MAT C = TRN(A)
  ```
  Transpose matrix A and store the resulting matrix in C.

- MAT vector multiplication statement

  ```
  MAT variable = V*W
  ```
  Multiply vectors V and W and assign the result to a variable.

- MAT zeros (0's) statement

  ```
  MAT C = ZER
  ```
  Set each element of matrix C to 0.

3.7.1. Matrix Dimensioning

An array variable used in a MAT statement should have its upper bounds (maximum) defined in a DIM statement.

For matrix operations, the lower bounds for each dimension of a matrix are assumed to be 1; elements in row and column 0 are unchanged.

Example:

```
100 DIM P(3,4)
```

This defines 20 elements P(0,0),...,P(3,4) but only 12 elements P(1,1),...,P(3,4) take part in any MAT operation.

The mathematical definition of matrix addition, subtraction, multiplication, inversion and transposition operations require the obvious conformities of matrix dimensions; otherwise, errors will result. Details concerning matrix dimensioning are discussed in the programming notes for each matrix operation statement.

Certain statements allow the user to implicitly or explicitly redimension a matrix. When a matrix is explicitly redimensioned, a trimmer is used that has a form similar to the array bounds listed in a DIM statement. Trimmers cannot change the number of subscripts of an array, but they can change the number of elements in the array (i.e., the user cannot change a matrix to a vector or vice versa).

When changing the number of elements in an array, the new array dimensions cannot cause it to have more elements than the original DIM statement reserved for it. If the original DIM statement reserved (n,m) elements, and the trimmer changes it to (a,b), the following condition must hold:

$$(a+1)*(b+1) \leq (n+1)*(m+1)$$

For example, if array A was dimensioned as 3,4 it could not be trimmed to 3,6, because the original matrix contained 20 elements and the new matrix would require 28 elements (remember row and column 0).

3.7.2. MAT Addition, Subtraction, and Multiplication Statements

These statements permit addition, subtraction, and multiplication of numeric matrixes.

Format:

```
MAT letter=letter+letter
MAT letter=letter-letter
MAT letter=letter*letter
```

Programming Notes:

1. The operator (+) denotes a matrix addition statement; the operator (-) denotes a matrix subtraction statement; and the operator (*) denotes a matrix multiplication statement.

2. Only one operation may be performed per statement.

3. Matrix dimensions must be conformable for each operation. If dimensions are not conformable, execution is terminated and a dimension error message is typed out at the terminal. The output matrix will be redimensioned, if possible, to be consistent with the input matrixes.

4. The following are treated as errors:

   MAT A=A*B
   MAT A=B*A

5. The mathematical definition of matrix multiplication is used. Thus, each of the following conditions must hold for MAT A=B*C:

   a. Current row bound (A) = current row bound (B)

   b. Current bound (A) = current column bound (C)

   c. Current bound (B) = current row bound (C)

   Matrix A will be redimensioned to meet these conditions.

   If either B or C is a vector, it will be transposed so that A will be a vector. If both B and C are vectors, an error will result. (See 3.7.13.)

6. The mathematical definition of matrix addition and subtraction is used. Thus, each of the following conditions must hold for MAT A=B+C or MAT A=B-C.

   a. Current row bound (A) $\geqslant$ current row bound (B)
      Current row bound (A) $\geqslant$ current row bound (C)

   b. Current column bound (A) $\geqslant$ current column bound (B)
      Current column bound (A) $\geqslant$ current column bound (C)

   Matrix A will be redimensioned to meet these conditions.

Example:

```
 10  DIM     A(2,2),B(2,2),C(2,2)
 20  FOR     I = 1 TO 2
 30  FOR     J = 1 TO 2
 40  READ    A(I,J),B(I,J)
 50  NEXT    J
 60  NEXT    I
 70  DATA    1,5
 71  DATA    2,6
 72  DATA    3,7
 73  DATA    4,8
 80  PRINT
 81  PRINT   ''MAT C = A + B''
 82  PRINT
 85  MAT     C = A + B
 86  GOSUB    200
 90  PRINT
 91  PRINT   ''MAT C = B - A''
 92  PRINT
 95  MAT     C = B - A
 96  GOSUB    200
100  PRINT
101  PRINT   ''MAT C = A * B''
102  PRINT
105  MAT     C = A * B
106  GOSUB    200
110  STOP
200  PRINT    A(1,1);A(1,2)
210  PRINT    A(2,1);A(2,2)
220  PRINT    B(1,1);B(1,2)
230  PRINT    B(2,1);B(2,2)
240  PRINT    C(1,1);C(1,2)
250  PRINT    C(2,1);C(2,2)
260  RETURN
300  END
```

The execution of the preceding program would produce the following output:

```
MAT C = A + B
 1   2
 3   4
 5   6
 7   8
 6   8
10   12
MAT C = B - A
 1   2
 3   4
 5   6
 7   8
 4   4
 4   4
MAT C = A * B
 1   2
 3   4
 5   6
 7   8
19   22
43   50
```

By using the MAT PRINT statement (3.7.9), statements 200 through 250 are replaced by

```
200 MAT PRINT A; B; C;
```

3.7.3. MAT Constant Statement

This statement results in all elements of the subject matrix being set to 1.

Format:

```
MAT letter = CON [(trimmer)]
```

where:

trimmer
: Is a new array dimension to be applied to the matrix.

Programming Notes:

1. A trimmer may optionally be used with this statement to dynamically redimension the matrix. This trimmer may not change the number of subscripts for the matrix. The new dimensions may not cause the new matrix to have more elements than did the original definition, or an error will result.

2. A trimmer has the same format as the dimensions on a DIM statement.

Example:

```
175 MAT C=CON
```

The elements of matrix C will be set to 1. The dimensions of matrix C are used in the operation.

3.7.4. MAT Identity Statement

The MAT identity statement is used to set the subject matrix to an identity matrix.

Format:

```
MAT letter=IDN [(trimmer)]
```

where:

trimmer
: Is a new array dimension to be applied to the matrix.

Programming Notes:

1. A trimmer may optionally be used with this statement to dynamically redimension the matrix. This trimmer may not change the number of subscripts for the matrix. The new dimensions may not cause the new matrix to have more elements than did the original definition, or an error results.

2. A trimmer has the same format as the dimensions on a DIM statement.

3. The current row and column dimensions of the subject matrix must be equal when this statement is executed; otherwise, an error occurs.

Example:

```
20   MAT B = IDN (3,3)
```

In the statement with line number 20, matrix B is changed to a 3 x 3 matrix and then set to an identity matrix. If B is not defined to be square, a dimension error message results.

3.7.5. MAT INPUT Statement

The MAT INPUT statement causes elements of the arrays in the array list to be assigned values during execution of the program. The terminal user is prompted by means of a question mark to enter a list of values. If the array name is not specified with a trimmer, or if the array name is not the last one in the list, the user must supply the same number of values as the current array dimension requires to fill the array. The user can always enter less than the required number of elements for the last array in the list. Therefore, the last array has a built-in trimmer feature. The number of values input is stored in the function NUM.

Format:

```
MAT INPUT mat-name [(trimmer)] [,mat-name[(trimmer)],...]
```

Programming Notes:

1. When the terminal user must enter an array in response to a MAT INPUT statement, it is quite likely that he will not be able to fit the entire array on a single line. The user may specify that a line is to be continued by entering a comma and an ampersand (&) following the last data item. The last line that is not terminated by an ampersand will terminate the input:

   ```
   ? 1, 2, 3, &
   4, 5
   ```

2. If the BASIC program is not doing vector input, then the number of data items typed must match the number of entries in the array.

3. When doing vector input, the vector is redimensioned to the number of values input, in addition to the value being stored in NUM. If the vector has a trimmer, however, NUM remains unchanged.

4. When inputting 2-dimensional arrays, elements in row 1 are filled first, then row 2, and so on.

5. NUM is updated only if the last variable in the list is a vector without a trimmer.

Example:

```
100 MAT INPUT A(3,4),V$
```

3.7.6. MAT Inversion Statement

Matrixes are inverted using the MAT inversion statement.

Format:

```
MAT letter=INV(letter)
```

Programming Notes:

1. Matrix inversion in place (MAT A=INV(A)) is treated as an error. If a matrix is singular, the value of the pseudo-function DET will be set to zero; otherwise, DET will contain the value of the determinant for the just-inverted matrix.

2. The mathematical definition of matrix inversion is used. Thus, each of the following conditions must hold for MAT A=INV(B):

   a. Current row bound (B) = current column bound (B)

   b. Current row bound (A) $\geqslant$ current row bound (B)

   c. Current column bound (A) $\geqslant$ current column bound (B)

3. The matrix being inverted is destroyed during the inversion process.

Example:

```
550   MAT K=INV(L)
```

Matrix K is made to represent an inverted row-column arrangement of matrix L.

3.7.7. MAT LINPUT Statement

This statement causes entire lines to be read into the elements of a string array during execution of the program. Matrixes are filled row-by-row until the entire matrix (except row and column 0) is filled.

Format:

```
MAT LINPUT string-array [(trimmer)] [,string-array [(trimmer)],...]
```

Example:

```
325 MAT LINPUT A$(5),C$
```

3.7.8. MAT Null Statement

This statement sets all elements of string matrix to null strings. The matrix may optionally be redimensioned.

Format:

```
MAT letter$ = NUL$ [(trimmer)]
```

where:

trimmer
: Is a new array dimension to be applied to the matrix.

3.7.9. MAT PRINT Statement

The MAT PRINT statement causes an entire array (except for row and column 0) to be printed row-by-row. If an array is followed by a semicolon separator, the elements of each row are printed closely packed; otherwise, the elements of each row are printed in columns 15 spaces wide. Each row begins on a new line. If a row does not fit on one line, it is continued on succeeding lines. If no print separator follows a vector, it is printed as a column vector, i.e., one element per line; otherwise, it is printed as a row vector.

Format:

```
MAT PRINT mat-name letter [{,} mat-name letter . . .] [,]
                          {;}                        [;]
```

where:

mat-name
: Is the name of a string or numeric matrix.

3.7.10. MAT READ Statement

The MAT READ statement causes elements of the matrixes in the array list to be assigned values during execution of the program. These values are obtained from the appropriate block data formed by the DATA statements. Matrixes are filled row-by-row until the entire matrix (except for row and column 0) is filled.

Format:

```
MAT READ mat-name [(trimmer)] [,mat-name [(trimmer)]...
```

where:

mat-name
: Is the name of a string or numeric matrix.

trimmer
: Is a new array dimension applied to the matrix.

3.7.11. MAT Scalar Multiply Statement

The expression is evaluated and this result is used to multiply each element in the matrix on the right side of the equal sign. The resultant values are assigned to the matrix on the left side of the equal sign.

Format:

```
MAT letter=(arithmetic-expression)*letter
```

Example:

```
190   MAT C = (5) * A
```

Each element in A is multiplied by 5 and the result is placed in matrix C. The dimensions of both matrixes must be identical.

3.7.12. MAT Transpose Statement

Matrixes are transposed using the MAT transpose statement.

Format:

```
MAT letter=TRN(letter)
```

Programming Notes:

1. Matrix transposition in place (MAT A=TRN(A)) is treated as an error.
2. The mathematical definition of matrix transposition is used. Thus, each of the following conditions must hold for MAT A=TRN(B):
   a. Current row bound (A) ⩾ current column bound (B)
   b. Current column bound (A) ⩾ current row bound (B)

Example:

```
300   MAT G=TRN(H)
```

The matrix G is the transpose of matrix H.

3.7.13. MAT Vector Multiplication Statement

This statement permits the multiplication of two vectors, yielding a scalar result.

Format:

```
MAT variable = letter * letter
```

Programming Notes:

1. Both arrays used in the statement must be defined to be vectors of equal size.
2. The result must be assigned to a numeric variable.
3. The variable must be in the form letter-number (such as A1 or B7) to explicitly denote a scalar variable.

Example:

```
MAT     A6 = V*W
```

3.7.14. MAT Zeros (0's) Statement

This statement results in all elements of the subject matrix being set to 0.

Format:

```
MAT letter=ZER [(trimmer)]
```

where:

trimmer
: Is a new array dimension to be applied to the matrix.

Programming Notes:

1. You can use this statement to dynamically redimension the matrix. This trimmer may not change the number of subscripts for the matrix. The new dimensions may not cause the new matrix to have more elements than did the original definition, or an error will result.

2. A trimmer has the same formats as the dimensions on a DIM statement.

Example:

```
150   MAT C = ZER(3)
```

The elements of matrix C are set to 0. The dimension of matrix C is changed to 3; then the operation is performed.

3.8. PROGRAM SEGMENTATION

The statements described in this subsection allow BASIC programs to be logically and physically segmented. The CHAIN statement allows a large program to be divided into several smaller ones that may be serially executed occupying the same main storage region. The CALL and SUB statements allow the development of parameterized, independent routines. The LIBRARY statement provides the mechanism for calling previously coded and debugged routines that have been stored in OS/3 library files.

CALL

3.8.1. CALL Statement

The CALL statement invokes a BASIC subroutine.

Format:

```
CALL string-constant[:param-list]
```

where:

`string-constant`
Is a subroutine name (eight alphanumeric characters maximum).

`param-list`
Is one of the following:
expression
variable
channel setter
function name
array

Five types of parameters may be specified in the param-list.

1. Expression (call-by-value) - Any numeric or string expression. The value is only passed to the subroutine; no value may be returned. A simple variable may be made an expression by enclosing it in parentheses.

   Example:

   A+3,5,(X),A$&B$,"ABC"

2. Variable (call-by-reference) - Any numeric or string variable. The value of the variable may be changed by the subroutine.

   Example:

   X,R3,A$,X$(1,3)

3. Channel setter - A file is passed to the subroutine. Any processing may be performed on the file by the subroutine, including reopening the file with a different name.

   Example:

   #1,#X+Y

4. Function name - A function is passed to the subroutine. The function may be used in any valid context in the subroutine. The number and type of parameters for the passed function must agree with its use in the subprogram.

   Example:

   ```
   FNX$,SIN
   ```

5. Array - An entire array may be passed to a subroutine. Any valid operation, including redimensioning, may be performed by the subroutine. Note that the CALL statement only specifies the number of dimensions, not the actual dimensions.

   Example:

   ```
   A(,),B$()
   ```

Programming Notes:

1. Subprograms may not be called recursively.
2. Only open files may be passed.
3. Arrays may be redimensioned in a subroutine by using them with trimmers.
4. Functions that are passed on CALL statements must be defined before the CALL statement.

Example:

```
100 CALL ''SUB1'':5+I,A$,#B,SIN B(,),''YES''
```

CHAIN

3.8.2. CHAIN Statement

This statement terminates the execution of the current program and initiates execution of a specified program. The chained program can reside in either an OS/3 library file or in a BASIC workspace file created by the chaining program. The CHAIN statement allows a large BASIC program to be segmented and new phases to be loaded without the terminal user being involved.

Format:

```
CHAIN  {#N                }  [WITH #I[,#J,...]]
       {string-expression}
```

where:

`#N`
Is a channel expression for a BASIC file containing a BASIC program.

`string-expression`
Is a program identifier of a BASIC program in an OS/3 library file. Its format is similar to that used on an OLD or RUNOLD statement.

`#I, #J,...`
Is a list of channel expressions specifying those files to be passed to the chained program. The passed files are assigned sequential channel numbers, beginning at 1. That is, in the chained program, the first file in the list is assigned to channel 1, the second to channel 2, etc.

Programming Notes:

1. If the chained program is specified by a channel expression, the file must be a temporary or library file; a MIRAM file is not permitted.

2. If the file containing the chained program is an OS/3 library file, the file will be closed after the chained program is loaded.

3. Any files not included in the file list are closed before the chained program is loaded.

4. The chained program source is not copied into the BASIC workspace. When execution of a chained program completes, the original contents of the workspace when the RUN or RUNOLD statement was issued is still intact.

Example:

```
900 CHAIN #3
950 CHAIN ''PHASE2,PROGLIB,PACK43'' WITH #10, #1
```

LIBRARY

3.8.3. LIBRARY Statement

This statement informs BASIC of the names of OS/3 library files that are to be searched to find subroutines referenced by the program.

Format:

```
LIBRARY file [(password)][,volume]
```

where:

`file`
Is the name of an OS/3 library file.

`password`
Is the READ password for the file. It must be included in the statement if the file has been cataloged with a password.

`volume`
Is the name of the disk pack on which the file resides. If the file has been cataloged with a volume name, this parameter may be omitted.

Programming Notes:

1. At load time, all subroutines in the program file are loaded first. Then, if there are unresolved subroutine names, the files specified in the LIBRARY statements are searched. If any subroutines are not resolved in this manner, execution is terminated.

2. A maximum of four LIBRARY statements are permitted in a BASIC program.

3. If more than one library is specified, the order in which they are searched is unpredictable.

4. In order for a subroutine to be found in a library, the SUB name must match the element name with which it was written to the library file.

5. Although multiple subroutines may be stored in the same library element, BASIC will only locate subroutines by the element name. Consequently, the element name must be the name of the first subroutine referenced in the program.

Example:

```
100 LIBRARY''SUBROUTINES(RDPASS),PACK33''
```

SUB

3.8.4. SUB Statement

This statement is the first statement of a BASIC subroutine. It must follow an END or SUBEND statement or be the first statement in a BASIC program file.

Format:

```
SUB string-constant [:param-list]
```

where:

`string-constant`
Is the subroutine name, consisting of no more than eight alphanumeric characters. If this subroutine is to be loaded implicitly by BASIC through the use of LIBRARY statements, this name must be the same as its element name in the OS/3 library file.

`param-list`
Is the list of local variables passed to the subroutine. Each must have the same type (string or numeric) and dimension (matrix, vector, scalar, function, or file) as the corresponding parameter in the CALL statement. These parameters may be:

variable
channel setter
function name
array

Four types of parameters may be specified in the param-list:

1. Variable - Any numeric or string variable. The corresponding CALL statement may contain a variable or an expression. When the caller passes a variable, subroutine references alter the value of that variable; when the caller passes an expression, the parameter is a local value. The subroutine is not aware of the different parameter modes. However, a returned value is lost if the subroutine is called with an expression.

2. Channel setter - Any channel constant (#1, #30, etc). References to this channel act upon the file passed by the caller. The file must be opened by the caller prior to calling the subroutine. Any files opened in the subroutine that are not included in the param-list will be local to the subroutine and will be closed upon exit.

3. FN letter [$] - Any user function may be defined in the SUB parameter list. Function result type and the types of each function parameter must be consistent with the function passed to the subroutine by the caller.

4. Array reference - Any array name may be defined here. The variable type and number of dimensions must be consistent with the passed arrays. No dimension statement for these arrays may appear in the subroutine. Note that no dimensions are included on the SUB line, only the number of dimensions.

   Example:

   ```
   A(,),X$()
   ```

Programming Notes:

1. Each SUB statement must define a unique subprogram name. Two or more subprograms with the same name in the user's program will result in an error.

2. Any variables, arrays, functions, or files not declared in the SUB line are local to the subprogram. Local arrays, functions, or files must be defined by the appropriate DIM, DEF, or FILE statement.

3. A SUB statement is only valid as the first statement in a library subprogram, or after an END or SUBEND statement.

4. Local variables contain unpredictable values when the subroutine is entered.

5. DATA statements are local to the subroutine. The DATA pointers are reset to the beginning of the data block on entry to the subroutine, and any READ statements issued within a subprogram will not interfere with READS or DATA in the calling program.

Example:

```
10000 SUB''SUB1'':X,Y$,#3,FNS,X(,)
```

SUBEND

3.8.5. SUBEND Statement

This statement is the last statement in a BASIC subroutine. If this statement is executed, control is returned to the caller.

Format:

```
SUBEND
```

Programming Notes:

1. The SUB and SUBEND statements delimit the subroutine. No statement within the subroutine may refer to a statement before the SUB or after the SUBEND.

2. If the subroutine is loaded from a LIBRARY statement, the line numbers within the subroutine are local to the subroutine and, in fact, may be duplicates of lines existing in the main program.

3.8.6. SUBEXIT Statement

The SUBEXIT statement terminates a subroutine and returns control to the caller. Unlike the SUBEND statement, the SUBEXIT may occur anywhere within the subroutine, except within a user-defined function.

Format:

```
SUBEXIT
```

Example:

```
983 SUBEXIT
```

CHANGE

3.9. CHANGE STATEMENT

The CHANGE statement converts arithmetic and alphanumeric formats. It can change a character string into an array of numeric values and vice versa.

Format:

```
CHANGE string TO array [BIT expr]
CHANGE array TO string-variable [BIT expr]
```

where:

string
Is any string expression, string variable, or closed string.

array
Is any numeric array name.

string-variable
Is the string variable that will contain the changed array.

expr
Is a numeric expression specifying the number of bits per character.

Programming Notes:

1. When changing from a string to a numeric vector, the BIT expression specifies the number of bits, n, which are used to form pseudo characters. The first n bits of the string are used to form a decimal number. This value is converted to floating point and stored in the first entry of the array. Then processing continues with the next n bits. If extra bits remain that would not complete a full character, they are ignored. The total number of entries converted is stored in the zero element of the vector.

2. When changing from a string to a vector, the vector must be large enough to accommodate all the character values or an error results.

3. When changing from a vector to a string, the user must set element 0 of the vector to the number of vector elements to be converted. Each element in the vector from the first to the last one the user selects is converted to a bit string of length n. These bit strings form the new string. If element 0 contains a 0, a null string is produced.

4. When changing from a vector to a string, if a converted element value cannot be represented in n bits or is negative, a runtime error results. Attempting to create a string greater than 4095 characters also results in an error.

5. If omitted, the BIT parameter defaults to eight. The maximum permissible value for the BIT expression is 24.

Examples:

```
100 CHANGE A$ TO X(15)
200 CHANGE Z TO B$ BIT 7
```

4. File Support

4.1. INTRODUCTION

The user can save file information permanently or retrieve it at any time using BASIC file capability. He can update file data, reference it in a program, or write new data to the end of a file. The type and format of these files are flexible, enabling the user to access files from both BASIC and batch programs.

4.2. FILE DESCRIPTION

Three file types are supported by BASIC: temporary files, library files, and MIRAM files. Although the file types may vary, the actual format of a data record processed by a given statement will not change. This allows a correctly written program to use the same statement to process a temporary, library, or MIRAM file interchangeably as long as the record content is the same.

- Temporary files

  These files are maintained entirely by BASIC and permit the user to create and read local files without the overhead of allocating space on the disk. When a FILE statement declares a temporary file, BASIC allocates one in its workspace. When the program or subprogram terminates, these files are erased.

- Library files

  Library files, or library elements, may be used for permanent storage of BASIC files. These files are stored as single librarian format elements within a SAT file, and may be accessed by the librarian, batch programs, and other system programs.

  Because library elements are sequential by nature and may not be extended or updated in place, they are copied to the BASIC workspace and accessed there. After the BASIC program has finished with the file (either at program or subprogram termination or when the file's channel number is reused by another FILE statement), the data is copied from the workspace back to the file and placed at the end, automatically deleting the old element if one exists. If no WRITE operations have taken place on the file, it will not be written back.

- MIRAM files

  Unlike library files, MIRAM files do not use the workspace; they process the data in place on disk. All types of MIRAM files (fixed length record, variable length record, keyed, and unkeyed) can be opened, but BASIC permits access to these files only by using the relative record number. They cannot be accessed by key. When the file is opened, its characteristics are obtained from the label (record size, buffer size, or file type). Although any type of MIRAM file may be accessed, all records written by BASIC will be unkeyed.

Any number of MIRAM files may be open simultaneously; however, no more than 32 library and workspace files may be open at the same time.

For a new file, BASIC will create a MIRAM file using the default MIRAM parameters. For additional information on MIRAM, refer to the consolidated data management concepts and facilities manual, UP-8825 (current version).

All BASIC files are controlled by several parameters defining which operations will be permissible for the file, and how the BASIC statements will operate. These parameters are the file type (library, temporary, or MIRAM), margin size, current location pointer, and end-of-file pointer. The file type is determined when the file is opened by the FILE statement. At the same time, a margin setting is determined which limits the maximum record size that can be written to the file. The current location pointer and end-of-file pointer are dynamic and change during execution. The current location pointer is initialized to zero and points to the next record to be read or written to the file at any given time. After a record is read or written, the pointer is advanced by one to point to the next record. At any time during execution, the user may change the current location pointer via a RESET statement; this will take effect on the next READ or WRITE. PRINT statements do not use the current location pointer, but always output records using the end-of-file pointer. This pointer is set to write records immediately following the last record in the file and is incremented once for each record written. The end-of-file pointer can only explicitly be reset by a SCRATCH statement, which erases the entire file contents and repositions both pointers to the start of the file.

Records in BASIC are numbered beginning with 0; the first record is at location 0, the second at location 1, and so on. The end-of-file pointer is always set to the last record in the file plus 1, so, if the file contains 105 records, the last record will be at location 104 and the end-of-file pointer will contain a value of 105.

BASIC files are composed of one or more records, with each record containing data in some user-defined format. Certain BASIC statements (such as INPUT) make assumptions as to the format of the data, and will scan off data from the records field by field. Other statements make no assumption as to the format, and allow the user to retrieve entire records and perform the field separation and conversion himself. When outputting records to the file, the user can format the entire record in a string variable and write it to the file (WRITE) or he can allow BASIC to perform the formatting and editing for him via the PRINT USING capability.

In general, field separation for file records follows the same rules as for data input from the terminal. On output, however, the user program must supply the separators that will be expected by BASIC when the file is read. When BASIC performs the field separation functions for the user, certain restrictions apply to the format of the data in the records. Numeric fields are composed of an optional sign, a series of digits with an optional decimal point, and an optional exponent field. The field must either terminate the record, or end with a comma. String fields may be either opened or closed, and must either terminate the record or end with a comma. Closed string must begin and end with a quote (") and must be the only data in the field. Quotes required within closed strings may be entered as two successive quote characters.

When numeric variables are read via the INPUT statement, the field used to supply the next value must be a numeric field or a fatal error will result. With string variables this is not a problem because the string contents may, in fact, be numeric digits.

The user must be aware of these restrictions if a file is to be created by BASIC and then read via INPUT statements; commas for field separators must be written explicitly to the file. For example, if a BASIC program would read data with the statement:

```
10 INPUT #3: A,B,C
```

the record would have to look similar to:

```
45.2, 45.6, 54.2
```

One statement to create this record could be:

```
23 PRINT #3: A1 ;'',''; B1 ;'','';C1
```

Note that because BASIC is performing field separation, and fields may either terminate the record or end with a comma, records to supply data for this INPUT could be any of the following examples:

```
45.2
45.6
54.2

45.2, 45.6
54.2

45.2, 45.6, 54.2, 64.7
```

In the last example, the value 64.7 would not have been read by the INPUT statement, but would be retained for the next INPUT (assuming the user does not reposition the file).

To uniquely identify each file, a channel number is required. The channel number to be used for a file is defined by the user in the FILE statement and must be in the range 0 to 4095. Once a file has been defined in the FILE statement, any future references to that channel number will initiate an access to that file. One special case of the channel number is channel 0, which is always defined to be the terminal. Statements such as PRINT, INPUT, and LINPUT may explicitly reference channel 0 to access the terminal, but normally no channel setter is specified because the statements default to the terminal.

4.3. FILE STATEMENTS

There are 10 BASIC statements and 5 matrix I/O statements used for files. A brief description of each file statement is shown in Table 4-1. These statements apply to all file types and perform the same function regardless of the file. This means that a program could be written with a sequential file in mind, but may also be used with a library file without program changes.

Table 4—1. BASIC File Statements (Part 1 of 2)

File Statement	Use
FILE	The FILE statement is used to declare a file and assign it to a channel number. This statement causes the file to be located on disk and opened for use. Once a file has been assigned to a channel number, any future references to that channel will refer to that file.
INPUT	One of the statements used to read data from a file is INPUT. Variables listed in the INPUT statement are filled by scanning values from the record. More than one value may be present in a record; each will be scanned off and assigned as needed to supply values for INPUT requests. Multiple data values on a single record must be separated by commas. Normally, records are read sequentially beginning with the first in order to obtain values for INPUT requests. The user, however, may change this by resetting the value contained in the current location pointer. This would cause a new record at the specified location in the file to be read to supply values for the next INPUT requests.

Table 4—1. BASIC File Statements (Part 2 of 2)

File Statement	Use
LINPUT	Entire records can be read into a single-string variable using the LINPUT statement. This enables the user to make use of the string and conversion functions in BASIC to strip off fields in the record when the format of the data values is not standard. As with the INPUT statement, LINPUT reads the file sequentially to fill the variables in the LINPUT list, but may be forced to begin reading records at a new location within the file by resetting the current location pointer.
MARGIN	All files in BASIC have a margin size that corresponds to the size of the largest record which may be written to that file. The default margin size for all files is 256 characters. The margin size will be set to the record size when a MIRAM file is opened. Most other files will receive the default margin setting. The MARGIN statement may be used to change the margin value during program execution.
Matrix I/O	When used with files, the matrix I/O statements may be used to perform selected operations on all elements of the matrix (except row and column 0). The user can use trimmers to dynamically change the array dimensions during execution.
PRINT	The PRINT statement may be used with files to write string or numeric data. Records written as a result of the PRINT statement are always appended to the file at the end, and the end-of-file pointer changed to show a longer file. Thus, PRINT corresponds to a sequential extension of the file.
READ	The READ statement is similar to the LINPUT statement, but may be used with string or numeric variables. When used with string variables, the statement functions identically to the LINPUT statement. When used with numeric variables, a record is read that is expected to contain a single numeric data item. This value will be converted to floating point and assigned to the numeric variable. As with the LINPUT statement, READ will access records sequentially unless the current location pointer is altered, in which case it will begin reading records at the new location.
RENAME	The RENAME statement provides the capability to change the name of an open file. When used with library files, BASIC discards the original name and notes the new name for use when the file is closed. MIRAM files may not be renamed. The RENAME statement may also be used with temporary files to change a temporary file to a library file (instead of scratching the file when it is closed, it will be written to a library), or a library file may be renamed to a temporary file (it will not be written back when closed, leaving the original copy intact). This facility may be used to create a new library element, by opening the file as a temporary file (*), and renaming it to a library element.
RESET	The RESET statement is used to reset the current location pointer in order to change the position in the file where INPUT, LINPUT, READ, and WRITE statements will operate. Certain restrictions apply to the use of RESET depending on the file type.
SCRATCH	The SCRATCH statement will erase the contents of a file. The file is not closed by this statement, so PRINT or WRITE statements may be used to write new data to the file. Note that when the file is scratched, the end-of-file pointer and current position pointer are both set to the beginning of the file.
WRITE	The WRITE statement is used to output variables, one per record, to the file. Either numeric or string variables may be used with the WRITE statement. When numeric values are written, they are converted to display format, padded with spaces if necessary to fill the record, and written at the current file pointer. The pointer is advanced once for each record written. String values are written in a similar manner to numeric values, except that no conversion is required.

4.3.1. FILE Statement

The FILE statement is used to assign a file to a channel number. The channel number must specify an integer value between 1 and 4095. The file name must be in a format compatible with the type of file being opened. The three types of files supported by BASIC (temporary, OS/3 library, and MIRAM files) are assigned using the FILE statement and either positional or keyword parameters.

- Positional parameters must be written in the order specified and must be separated by commas. When a positional parameter is omitted, the comma must be retained to indicate the omission, except for the case of omitted trailing parameters.

- A keyword parameter consists of a word or a code immediately followed by an equal sign, which is, in turn, followed by a specification. Keyword parameters can be written in any order. Commas are required only to separate parameters.

If a previous file had been assigned to the same channel number, that file is closed before the new one is opened.

Format:

```
FILE channel-setter: ''string-expression''
```

where:

`channel-setter`
Identifies the channel number assigned to the file. All future references to the file use this number.

`''string-expression''`
Is a string expression identifying the file that is being opened. Its exact format varies with the different types of files available. The string-expression must be enclosed in "(double quotes).

NOTE:

Shaded areas in the following formats indicate the default; underlined letters indicate that the system will accept a portion of the keyword.

Temporary file format:

```
filename
```

where:

`filename`
Must be specified with an asterisk (*).

OS/3 library file positional parameter format:

```
modulename,filename[(readpassword/writepassword)][,volume][,module-type]
```

OS/3 library file keyword parameter format:

```
MODULE=modulename ,FILENAME={filename     }[,RDPASS=readpassword]
                            {'filename'   }
                            {''filename''}
      [,WRPASS=writepassword][,VSN=volume][,DEVICE={addr    }]
                                                   {DISK    }
                                                   {DISKETTE}
      [,TYPE=module-type]
```

where:

MODULE=modulename
: Specifies the name of the module referenced. It can be one to eight alphanumeric characters.

FILENAME={filename / 'filename' / ''filename''}
: Specifies the name of the library file being referenced. The physical file names may be 1 to 44 alphanumeric characters. If there are spaces, commas, or parentheses embedded in the file name, it must be enclosed in either quotation marks or apostrophes.

RDPASS=readpassword
: Specifies a password used to control the read access to a file being referenced. A password is required if the file is listed with a password in the file catalog. If the file is to be cataloged and file protection is desired, the user must specify passwords. Passwords may be one to six alphanumeric characters.

WRPASS=writepassword
: Specifies a password needed to control the write access to the file being referenced. A pasword is required if the file is listed with a password in the file catalog. If the file is to be cataloged and file protection is desired, the user must specify passwords. Passwords may be one to six alphanumeric characters.

VSN=volume
: Specifies the volume serial number indicating the volume on which the file the user wants to access resides. The volume serial number is required if the file is not cataloged and may be one to six alphanumeric characters.

DEVICE={addr / DISK / DISKETTE}
: Specifies the type of device read from or written to by the user. addr specifies a 3-digit hexadecimal number indicating the physical device address of the device the user wants to use. The first digit is the channel number, the second the control unit address, and the third the device number. If a device is not specified, the parameter will default to DISK.

TYPE=module-type

Specifies the type of module being referenced. The module-type is indicated by entering a letter corresponding to the module-type the user wants to reference. For SAT files, the types permitted are: source S, macro M, procedure P, load L, and object O. For MIRAM files, specify format F, saved job control stream J, or one of many other types. The user may also create his own module-types and identify them with a 1- to 4-character name. A module-type can serve as a qualifier for a module. The default is S.

OS/3 MIRAM file positional parameter format:

```
'filename'[(readpassword/writepassword)][,volume]
```

OS/3 MIRAM file keyword parameter format:

```
FILENAME={filename     }[,RDPASS=readpassword][,WRPASS=writepassword]
         {'filename'   }
         {''filename'' }
        [,VSN=volume][,DEVICE={addr     }][,INIT={YES}][,RCSZ={record size}]
                              {DISK     }       {NO }        {256        }
                              {DISKETTE }
```

where:

```
FILENAME={filename     }
         {'filename'   }
         {''filename'' }
```

Specifies the name of the MIRAM file being referenced. The physical file name may be 1 to 44 alphanumeric characters. If there are spaces, commas, or parentheses embedded in the file name, it must be enclosed in either quotation marks or apostrophes.

RDPASS=readpassword

Specifies the read access to a file being referenced. A password is required if the file is listed with a password in the file catalog. If the file is to be cataloged and file protection is desired, the user must specify passwords. Passwords may be one to six alphanumeric characters.

WRPASS=writepassword

Specifies a password needed to control access to the file being referenced. A password is required if the file is listed with a password in the file catalog. If the file is to be cataloged and file protection is desired, the user must specify passwords. Passwords may be one to six alphanumeric characters.

VSN=volume

Specifies the volume serial number indicating the volume on which the file the user wants to access resides. The volume serial number is required if the file is not cataloged and may be one to six alphanumeric characters.

```
DEVICE={addr     }
       {DISK     }
       {DISKETTE }
```

Specifies the type of device read from or written to by the user. addr specifies a 3-digit hexadecimal number indicating the physical device address of the device the user wants to use. The first digit is the channel number, the second the control unit address, and the third the device number. If you do not specify a device, the parameter will default to DISK.

INIT={YES / NO}

Specifies the overwriting of the contents of a file with new data. If YES is specified, whatever data is presently in the file will be overwritten and a new file started. If NO is specified, the old data will remain intact and new data will be added to the end of the file. If neither YES or NO is specified, the parameter defaults to NO.

RCSZ={record size / 256}

Specifies the size of the record that BASIC is to process. This parameter incorporates the margin size statement in that the following are identical:

```
100 FILE #1: ",MIRAMFILE,PACK"
200 MARGIN #1: 512
```

is equivalent to:

```
100 FILE #1: ",MIRAMFILE,PACK,RCSZ=512"
```

The default value is 256 characters.

Programming Notes:

1. The FILE statement opens a BASIC file. Files are closed when a second FILE statement is issued for the same channel number or when the program terminates. Local files opened by subprograms are closed when the subprogram terminates (SUBEXIT or SUBEND).

2. If the file name specifies an asterisk (*), then the file is a temporary file maintained by BASIC in its workspace. The file is scratched when it is closed.

3. If the file name specifies an OS/3 library file, the file is copied to the BASIC workspace when it is opened. Once in the workspace, the file is identical to a temporary file except that it will be copied back to the library when it is closed. The library file must be copied because the format of an OS/3 library file does not permit updating records in place or extending an element.

4. If the file name specifies a MIRAM file, the file will not be copied; BASIC processes these files in place. When the file is opened, its characteristics will be obtained from the VTOC. These will determine the record size (MARGIN) and types of access permitted.

5. BASIC processes MIRAM files and permits access to these files only by using relative record number.

6. BASIC processes files with record sizes up to 16K bytes and buffer sizes up to 32K bytes. Within these limits, any record sizes and buffer sizes are permitted.

7. MIRAM files must exist before they can be opened by a FILE statement. If, upon opening a file, it is found to be empty, the default margin size is taken (256), the record and buffer sizes are set to the margin size, and the file is assumed to have fixed length records. This is the BASIC default file specification.

8. A library element must exist before it can be accessed by a FILE statement. If a new element is to be created as a BASIC file, it should be built as a temporary file with a margin not greater than 256 characters, and changed to a library element prior to being closed with the RENAME statement. (See 4.3.8.)

9. If the file has been password protected, the correct passwords must be entered in the FILE statement. Failure to enter the READ password (if required by the catalog) will inhibit any READ operations. Failure to correctly enter the WRITE password (if required) will inhibit any WRITE operations. If a file has both the READ and WRITE passwords cataloged and neither is specified by the user, access to the file will be denied (the program could not do anything regardless because both READ and WRITE would be inhibited).

Examples:

```
100 FILE #F9:F9$
200 FILE #1:''DATA,BASICLIB,DISKØ3''
300 FILE #4000:'''*'''
400 FILE #D:'''MIRFILE',DISKØ1''
500 FILE #1Ø:N1$&'',LIBRARY,PACKØ2''
600 FILE #47:''PAYROLL''(A234/A432)''
```

INPUT

4.3.2. INPUT Statement

The INPUT statement allows the user to read a list of values from a record in the file. These values must be formatted in the record just as they have to be formatted if entered at the terminal as an INPUT response. If there are insufficient values on a given record, BASIC continues reading records until it has filled all of the variables in the program's "input list". Unlike input from the terminal, there is no relationship between the structure of the INPUT statements and the records in the file. Thus, example 1 and example 2 are functionally identical.

Data items read by INPUT statements are taken from fields within the records and may be numbers, open strings, or closed strings. If the wrong type of data is supplied for a variable in the input list, a fatal error will result. When strings are read in, leading and trailing spaces are deleted unless the string in the field is enclosed in quotes. When quotes are used, the characters within the quotes are assigned without any editing. Note that to output quotes to a record they must be explicitly printed as in example 3.

Format:

```
INPUT channel-setter : variable[,variable...]
```

where:

`channel-setter`
Selects the file to be read.

`variable`
Is a numeric or string variable or array element.

Programming Notes:

1. Records required by INPUT requests are retrieved sequentially beginning with the first record in the file. The current location pointer is incremented immediately when a record is read, not when all fields in the record have been processed. The RESET statement may be used to change the location where the next record will be read.

2. More than one data field is permitted on a single record. If an INPUT statement does not exhaust all fields in a record, the remaining fields are retained for subsequent INPUT statements. The remaining fields will be lost if output is written to the file or the current location pointer is changed; subsequent INPUT statements will force a new record to be read.

3. Numeric data fields contain leading or trailing spaces, must contain a valid number, and must end with a comma or be the last field in the record. It is not an error to supply a numeric data field to a string variable on INPUT; the character string consisting of the numeric digits will be used.

4. String data fields may be open or closed strings. Open-string fields may contain any valid characters and terminate with a comma or at the end of the record. Closed strings must begin and end with quotes ("). Leading spaces before the first quote are permitted, as are trailing spaces between the last quote and the comma or end of record. A fatal error will result if a string data field is supplied for a numeric variable.

Example 1:

```
100 INPUT #1:A,B(5),C$
```

Example 2:

```
100 INPUT #1:A
101 INPUT #1:B(5)
102 INPUT #1:C$
```

Example 3:

```
100 LET A$='' '' '' ''(or CHR$(EBC(''')))
110 PRINT #124:A$ &''ABC'' & A$
120 RESET #124: LOF(#124)-1
130 INPUT #124: R3$
```

This example writes a record containing

```
''ABC''
```

to the file. Statement 120 repositions the current location pointer to the end-of-file record minus one, which is the new record. This value can then be read into variable R3 without losing any spaces that may be significant. It is important to note that statement 110 was not coded as

```
110 PRINT #124: ;A$ ;''ABC'';A$
```

because it is possible (although unlikely) that one of the three fields in the second format could fill the record and, thus, two records could be printed:

```
''ABC
''
```

Concatenating all three fields ensures that they will be printed as one string.

LINPUT

4.3.3. LINPUT Statement

The LINPUT statement allows the user to read in entire records; each record is read into a single string variable. Because the record contents are ignored when this assignment is made, any data may be read into a string from the file. This permits the user to read a record and strip off fields via the string functions in cases where an INPUT statement would not find the data in the correct format. Completely blank records are permitted and are stored in the string variable as null strings.

Format:

```
LINPUT channel-setter:string-variable[,string-variable...]
```

where:

channel-setter
Selects the file to be read.

string-variable
Is a string variable or string array element where the record contents are to be stored.

Programming Notes:

1. If the last statement issued to the file was an INPUT and there is still data in the record which has not been read, LINPUT will use the remaining characters in the record instead of requesting a new record. The next variable to use LINPUT will then force a record to be read.

2. If the last statement issued to the file was other than an INPUT, or if it was an INPUT and there is no data remaining in the record, a new record will be read for the string variable.

3. Records required for LINPUT requests are retrieved sequentially beginning with the record at the current location pointer and the pointer is incremented for each record read. In other words, the record is incremented once for each variable in the LINPUT list. The RESET statement may be used to alter the location where the next LINPUT will begin retrieving records.

4. Leading spaces in records are not removed. Trailing spaces are eliminated.

Examples:

```
940 LINPUT #1:A$
950 LINPUT #1:B1$ , C$(3,4)
960 LINPUT #4:D$(E+1)
```

4.3.4. MARGIN Statement

The MARGIN statement permits the user to change the current margin setting for a file. The initial margin setting is determined when the file is opened. For temporary and library files the default margin is used (256 characters). Existing MIRAM files acquire a margin setting from the maximum record size specification stored in the VTOC entry for the file.

Format:

```
MARGIN channel-setter:expression
```

where:

channel-setter
Identifies the channel number of the file to be altered.

expression
This value will be truncated to an integer value and used as the new margin setting.

Programming Notes:

1. The current margin setting limits the maximum record size that may be written to the file. Any attempt to exceed this limit will cause an error.

2. If the margin is changed while there is a record waiting to be completed (as a result of a PRINT statement ending with a comma, for example), the record being formatted will be written out prior to changing the margin.

3. The margin expression must result in a number between 1 and the following limits:

Temporary files	496 characters
Library files	256 characters
MIRAM files	16K characters

4. A temporary file or library file receives a default margin specification of 256 characters, which may be changed at any time after the FILE statement has been issued.

5. The margin size for a MIRAM file may only be changed when the file is empty and no data records have been formatted. This condition occurs if an empty file is opened, or immediately after a file has been scratched.

6. When the MARGIN statement is used on a MIRAM file, the string expression in the FILE statement must not contain any blanks.

7. The MARGIN statement and the RCSZ FILE statement parameter can't be used together on the same file. For example:

```
10 FILE #1:'',MIRAMFILE,VOLUME,INIT=YES,RCSZ=512 ''
```

is the same as:

```
10 FILE #1:'',MIRAMFILE,VOLUME''
15 MARGIN #1:512
```

Examples:

```
10 MARGIN #3:80
20 MARGIN #1:20*W
```

4.3.5. Matrix I/O Statements

To simplify the handling of matrixes when they are used with files, five matrix I/O statements are provided in BASIC. These statements perform the selected operation on all elements of the matrix except those in row and column 0. Processing for vectors begins with element 1 and continues to the last element in the vector. Arrays are processed beginning with element 1,1, then 1,2, continuing to 1,n, then row 2, row 3, and so on.

Supported statements include matrix PRINT, INPUT, LINPUT, READ, and WRITE. In general, the statements work just as if each matrix element were coded in the statement. For example:

```
MAT PRINT #3:A;
```

is interpreted as:

```
PRINT #3: A(1,1);A(1,2);A(1,3);...;A(1,n);
PRINT #3: A(2,1);A(2,2);A(2,3);...;A(2,n);
                .
                .
                .
PRINT #3: A(m,1);A(m,2);A(m,3);...;A(m,n)
```

Trimmers, when used, dynamically change the array dimensions during execution. This change is made just prior to performing the indicated file operation.

Formats:

```
MAT PRINT channel-setter:matrix[separator[matrix]],...
MAT INPUT channel-setter:matrix[(trimmer)],...
MAT LINPUT channel-setter:string-matrix[(trimmer)],...
MAT READ channel-setter:matrix[(trimmer)],...
MAT WRITE channel-setter:matrix,...
```

where:

`channel-setter`
: Selects the previously opened file for the indicated file operation. Channel 0, the terminal, may not be specified for MAT READ or MAT WRITE.

`matrix`
: Is a string or numeric matrix name.

`string-matrix`
: Is the name of a string matrix. Numeric matrixes are not permitted with this statement.

`separator`
: Is a PRINT item separator, such as a comma or semicolon, and determines the spacing of the printed elements in the record.

`trimmer`
: Is an optional matrix trimmer expression. This specifies the new matrix dimensions to be applied before the indicated operation is performed.

Programming Notes:

1. A trimmer may be used with the MAT INPUT, LINPUT, or READ statements to dynamically redimension the matrix. This trimmer may not change the number of subscripts for the matrix. The new dimension may not cause the new matrix to have more elements than did the original definition, or an error will result.

2. The MAT PRINT statement for files uses commas and semicolons to control spacing of elements in the records. If the matrix name is followed by a semicolon, the elements are printed closely packed. A comma following the matrix name causes the elements to be printed in 15-character columns. Each row begins a new line. If no print separator follows a vector, it is written as a column vector, one element per record; otherwise, it is printed as a row vector. The rules used in printing records to files are defined in 4.3.6.

3. The MAT INPUT, LINPUT, READ, and WRITE statements for files perform the indicated operation once for each element in the matrix. The rules covering the file INPUT, LINPUT, READ, and WRITE statements are defined in 4.3.2, 4.3.3, 4.3.7, and 4.3.11, respectively.

Examples:

```
120 DIM A(7),C$(3,5),D(2,8),E$(5),K(9),J$(4),K$(2,4),R(20)
122 MAT PRINT #3:A,E$;C$
123 MAT PRINT #3:D;
124 MAT READ #3:R,E$
125 MAT INPUT #4:K,E$(J)
126 MAT LINPUT #78:K$,J$
127 MAT WRITE #1:K$,R,A
128 MAT READ #1+1:K$(3),A(3)
129 MAT LINPUT #41:D(2,1)
```

PRINT

4.3.6. PRINT Statement

The PRINT statement may be used with any file accessible under BASIC to format all or portions of a record. The list of variables specified on the statement are written one after the other according to the print separators used between each item in the print list.

Any records written with a PRINT statement are always appended to the end of the file (the file will get longer). As each record is written, the end-of-file pointer is incremented by one to allow reading of all records up to and including the newly printed one. Resetting the current location pointer has no effect on the PRINT statement.

The PRINT statement is also affected by the MARGIN setting. If the user attempts to print more data in a single record than the margin will allow, BASIC then prints as many fields as it can on the first record and continues on a second record. No single data item longer than the margin setting can be printed.

Format:

```
PRINT channel-setter:[item[separator[item]]...]
```

where:

`channel-setter`
: Is the file to which this record will be written.

`item`
: Is an expression or a TAB reference.

`separator`
: Is a comma (,) or a semicolon(;).

Programming Notes:

1. Print separators may be used to control horizontal positioning within a record. If a semicolon is used after an item, the next item will be printed beginning at the next position in the record. If a comma is used, the next item will be printed beginning at the next 15-character field in the record (the record is broken into fields of 15 characters each and the next free field is used). If there is insufficient space in the current record, it is written out and a new record begun.

2. The TAB function may be used to advance to a specific position in the record. If the direction of the TAB is backwards, the current record is written out and a new record begun. The function of the comma and semicolon remains unchanged.

3. Null strings cause no data to be written to the record.

4. Numeric data is formatted either as an integer or decimal number. An integer number will be printed as an integer. A decimal number will be printed without the exponent field whenever possible. In either case, no more than six significant digits will be printed and a space will follow every number printed. If the number is positive, the sign is not printed but its print position is left blank; otherwise, a minus sign is printed.

5. If the statement ends with a separator, the record will not be written immediately, but will be held until another PRINT statement completes the record, or any other statement references the file.

6. If there are no items included in the list, the PRINT command will serve to write a previously unprinted record, or to print a blank record if the buffer is empty.

READ

4.3.7. READ Statement

The READ statement is somewhat similar in function to the LINPUT, in that there is a one-for-one correspondence between variables in the statement and records in the file, except that both string and numeric variables are permitted. When used with string variables, READ will retrieve a record and assign its contents without editing to the variable. When a numeric variable is specified, a record is read that must contain a single numeric value. This value is converted to floating point and stored in the variable.

Format:

```
READ channel-setter : variable-name[,variable-name...]
```

where:

`channel-setter`
Specifies the channel number of an open file to be read. Channel 0, the terminal, may not be referenced by this statement.

`variable-name`
Is a string or numeric variable or array element into which the data is to be read.

Programming Notes:

1. One record is read beginning at the current location pointer for each variable in the list. For each record read, the current location pointer is incremented by one. Changing the current location pointer via a RESET will select the location of the next record to be read by the READ statement.

2. READ does not check if the last operation on the file was an INPUT (as LINPUT would), but always reads new records.

3. When reading string variables, the entire record including any leading spaces is assigned without editing to the variable. Trailing spaces in the record will be eliminated.

4. When reading numeric variables, the entire record may contain only a single number; it will be converted and assigned to the variable. If the record contains any data other than a single number an error occurs.

Examples:

```
43 READ #43: A$,B4$,C$(H)
44 READ #44:A,B7,C(I,J)
45 READ #37:D,E8$
```

4.3.8. RENAME Statement

The RENAME statement will change the name of a BASIC file while it is contained in the workspace. In particular, it permits a library file element to be copied or created.

Format:

```
RENAME channel-setter:file-name
```

where:

channel-setter
Is a channel expression identifying an open file which is to be renamed.

file-name
Is a string expression specifying an OS/3 library file or a work file. Its format is similar to the file name used with a FILE statement.

Programming Notes:

1. Permanent MIRAM files may not be renamed. An attempt to do so will terminate execution of the program.

2. A temporary file may be renamed to a library file in order to create a new element in a library file.

3. A library file may be renamed to a temporary file in order to prevent the original copy of the file from being updated when the file is closed.

4. If the programmer wants to ensure that a file is not updated unless a specific condition occurs first, he should open the library file and immediately rename it as a temporary file. Then, if an error should occur during processing or if the terminal user should terminate the program, the library file will not be updated. Once the program has determined that the file is complete, the file can be renamed to a library file so that when it is closed, the library file will be updated.

5. If a program opens a library file with name A, processes the file, renames it B, and then terminates, the original copy of A will not be modified and a new modified version will exist with the name B.

Examples:

```
1Ø45 RENAME #1:''*''
2Ø74 RENAME #N:''NEW,LIBRARY,PACKØ9''
```

RESET

4.3.9. RESET Statement

The RESET statement is used to reposition the current location pointer to any location within the file. The statement may be used with or without a record number. When the record number is omitted, RESET goes to the beginning of the file – record 0.

Format:

```
RESET channel-setter[:numeric-expression]
```

where:

channel-setter
Selects the file to be repositioned.

numeric-expression
Is the new location of the file.

Programming Notes:

1. The numeric expression, if present, must result in a nonnegative number and the new location must not be greater than the current value of the end-of-file pointer.

2. A RESET statement without a record number is permitted to position any file type to the start of the file.

3. A RESET statement with a record number can be used with temporary or library files, or with MIRAM files.

Example:

```
35 RESET #4
```

4.3.10. SCRATCH Statement

The SCRATCH statement is used to erase the entire contents of a file. If the file is a temporary or library file, the scratch will only operate upon the workspace; the library file itself will not be affected. If the file is a MIRAM file, the scratch will erase the contents of the file. If there is no subsequent operation to the file, the file will be scratched from the disk.

Format:

```
SCRATCH channel-setter
```

Programming Notes:

1. If a MIRAM file is to be physically scratched from the disk, the SCRATCH operation should be the last operation issued against the file by the BASIC program.

2. If a MIRAM file is to be rewritten from the beginning, then the SCRATCH operation should be issued prior to writing to the file.

3. After a SCRATCH command, both the LOC and LOF of the file will be 0.

4. SCRATCH currently has no effect for library files.

Example:

```
10 SCRATCH #6
```

WRITE

4.3.11. WRITE Statement

The WRITE statement writes a list of variables to the file, one value per record. String text is written without any editing other than space filling if necessary. Numeric values are converted to display format and padded with spaces to fill the record. Depending on the position of the current location pointer, records are either updated or appended to the end of the file.

Format:

```
WRITE channel-setter:expression[,expression...]
```

where:

`channel-setter`
Specifies the channel number of the open file to which records are to be written. Channel 0, the terminal, may not be referenced by this statement.

`expression`
Is either a string or numeric expression to be written to a record in the file.

Programming Notes:

1. Each variable occupies one record, which is written at the position in the file specified by the current location pointer. After each record is written, the pointer is incremented. The RESET statement may be used to set the location where records will be written.

2. If the current location pointer is set to the end-of-file value, a new record will be added to the file and the end-of-file pointer advanced. If the current location pointer is set less than the end-of-file value, the record which was there will be overlaid by the new record, creating an update. The current location pointer may not be set past the end-of-file pointer.

3. Data to be written to the file may not be greater in length than the current margin setting for the file.

4. The WRITE statement may be used with temporary files, library files, and MIRAM files.

Examples:

```
8710 WRITE #10:''RECORD ONE'',''RECORD TWO''
8720 WRITE #10:3,4,Q5
8730 WRITE #10:A+6,B$,C4$(8),SEG$(D$,1,9)
```

5. BASIC Commands

5.1. INTRODUCTION

This section contains a detailed description of the operation and editing commands provided by the BASIC system. These commands enable the programmer to assign a name to a program, execute a program, and return control to the system. Editing commands are distinguished from source statements by the absence of prefixed line numbers. Once entered into the BASIC system, the editing command operates immediately on the current contents of the user's workspace, which can contain either a new program (being constructed) or a saved program.

The editing commands provided by BASIC provide the ability to enter, delete, list, and modify text on a single or multiple line basis. When extensive modifications must be made, the user should consider using EDT.

5.1.1. Definitions

The following syntactic units occur several times in the specification of the editing commands:

- line-number: a series of digits in the range of 1 to 99999

```
list-items: { line-number { , line-number [ , list-items ] }
            {             { , list-items                  } }
list-items: { program-name , file-name [ ( password ) [ , volume ] ] }
            {                          [ , volume                   ] }
```

- search-string: "characters"

Programming Notes:

1. Letter and digit are defined in Section 2.

2. A program name may contain from one to eight letters or digits, the first of which is a letter. Embedded characters such as $, ?, #, @, %, and hyphen may be included in this program name.

3. A file name may be up to 44 characters long. The same character construction rules which apply to program names also apply to file names.

4. A password may be up to eight characters long. The same character construction rules which apply to program names also apply to passwords. A password may be required if the file specified by file name has been cataloged with a password. When reading from a file (OLD), the read password may be required. When writing to a file (SAVE), the write password may be required. If the file is not cataloged, or no password is listed in the catalog, then the user's password specification, if any, is ignored.

5. A volume must be six characters and is made up of letters and digits. This name is used to locate the disk on which the referenced file exists. If the file is cataloged and a volume name has been listed, then the user may omit the volume entry. In any case, if a volume is listed, this overrides the catalog volume name.

6. All library file references refer to source elements, which may have been created by the OS/3 MIRAM librarian (MLIB), OS/3 general editor (EDT), or OS/3 BASIC.

7. All references to *the system* apply to the OS/3 BASIC System.

8. A search string is constructed in the same way as a closed string, and allows the user to selectively process source statements based on their content.

5.2. COMMANDS

The editing commands available to the user are given as follows:

```
BYE
DELETE
HELP
LIST
MERGE
MODIFY
NEW
OLD
PRINT
RESEQUENCE
RUN
RUNOLD
SAVE
SYSTEM
```

5.2.1. BYE

The BYE command is used to terminate BASIC. Control is returned to the system. All workspace information is lost.

The following message is displayed on the terminal:

```
BA113 BASIC TASK NORMAL TERMINATION
```

The BYE command does not return a response.

Format:

```
BYE
```

DELETE

5.2.2. DELETE

This command may be used to delete one or more lines of source from the user's workspace. If no line numbers are specified, the entire program is cleared.

Format:

```
DELETE    [list-items]    [search-string]
```

Note that single lines may be deleted by typing the line number of the line to be deleted. If a search string is specified, then the selected lines will be searched and those containing the string will be deleted.

5.2.3. HELP

Additional information about a status or error condition may be obtained by using the HELP command. Several lines of explanation will be displayed at the terminal. If a message number is not specified, BASIC displays a HELP message for the latest input syntax error.

Format:

```
HELP      [message-number]
```

LIST

5.2.4. LIST

The LIST command directs the system to display on the user's terminal the lines or sequence of lines referenced in the user's workspace. If no line numbers are specified, all statements in the program will be printed. If a search string is specified, then the selected lines will be searched and those containing the string will be printed.

Format:

```
LIST    [list-items]    [search-string]
```

5.2.5. MERGE

The MERGE command allows the contents of a library file to be added to the current contents of the workspace. Its function is identical to that of the OLD command, except that the workspace is not erased first.

Format:

```
MERGE     file-parameters
```

Programming Note:

If lines are read that duplicate the line numbers of lines already in the workspace, the new lines replace the old.

MODIFY

5.2.6. MODIFY

This format is used to correct or reenter a source statement from the terminal. The format is entered as if a new statement is being input. Any statement with the same line number is deleted and the new statement is substituted in its place.

Format:

```
line-number     statement
```

5.2.7. NEW

The NEW command erases the current contents of the user's workspace. BASIC will then respond with an asterisk. BASIC is now in the same condition it would be in if the user had just executed it from the system.

Format:

```
NEW
```

OLD

5.2.8. OLD

The OLD command erases the current contents of the user's workspace, then locates and loads the specified program into the user's workspace.

Format:

```
OLD     file-parameters
```

Programming Notes:

1. Errors may occur when BASIC is trying to locate the program if the disk volume, disk file, or element cannot be found. Errors may also occur if a password is required but not specified in the command.

2. As statements are read from the library file, each is verified by the syntax checker. Any statements in error are displayed on the user's terminal, and are entered into the program file, with a notation that the line is not valid. This permits the LIST command to show these lines so the user may later correct them.

3. Once all statements have been processed, control is returned to the terminal where new statements may be added, corrections made, or editing commands entered.

4. If a RUN is issued while there are still uncorrected lines from a previous OLD command, the lines which are in error will be rejected.

5.2.9. PRINT

The PRINT command directs the system to display on the user's terminal the lines or sequence of lines referenced in the user's workspace. If no line numbers are specified, all statements in the program will be printed. If a search string is specified, then the selected lines will be searched and those containing the string will be printed.

Format:

```
PRINT [list-items] [search-string]
```

RESEQUENCE

5.2.10. RESEQUENCE

This command will resequence a BASIC program. Because resequence is a complex operation requiring two passes over the source file, it is combined with a SAVE operation and may only be used with a syntactically correct program.

Format:

```
RESEQUENCE [start] [:increment] [:file parameters]
```

Example:

```
RESEQUENCE      100:50:MYPROG,MYFILE,MYPACK
```

Programming Notes:

1. If omitted, the starting value and increment default to 100.
2. The resequence operation will not be completed if the new highest line number would be greater than 99999 or if any line contains a syntax error.
3. An error will occur if any line of text must be expanded beyond 80 characters in order to insert the new line numbers.
4. The contents of the workspace are not modified.

5.2.11. RUN

The RUN command directs the system to load and execute the program contained in the user's workspace.

Format:

```
RUN
```

RUNOLD

5.2.12. RUNOLD

The RUNOLD command combines the functions of the OLD command and the RUN command. It eliminates the time-consuming step of writing the program into the workspace. Consequently, the source code is not available for editing. Statements are read from the library file, compiled, and written directly into main storage. Because this command is intended to be used to execute debugged programs, statement numbers are discarded to conserve main storage.

Format:

```
RUNOLD file-parameters
```

Programming Notes:

1. Errors may occur when BASIC is trying to locate the program if the file parameters are not correct.

2. If there are any syntax errors detected, the command will be terminated. The program will not be in the workspace, and an OLD command will have to be issued before the program can be corrected.

3. If execution errors occur, line number 0 will be displayed as the error location, because line numbers are not saved during RUNOLD processing.

5.2.13. SAVE

The SAVE command directs the system to save, on a OS/3 library file, a copy of the source program currently contained in its workspace. The program name is entered in the file directory and the body of text is stored as a source element. This element may later be retrieved using the OS/3 librarian LIBS, the OS/3 EDT program or OS/3 BASIC.

Format:

```
SAVE      file-parameters
```

Programming Notes:

1. Errors may occur if the disk volume or disk file cannot be located, or if a password is required but not specified in the command.

2. If a program with the same name already exists in the file, BASIC will ask:

   ```
   IS100 FILE/MODULE ALREADY EXISTS; OK TO WRITE:IT? (Y,N)
   ```

   A response of Y will delete the old copy and overwrite it with the new program.

   A response of N will terminate the command immediately and will leave the old copy of the program intact.

SYSTEM

5.2.14. SYSTEM

This command serves the dual purpose of breaking into system mode without destroying the contents of the workspace (compare with BYE) and of providing the ability to execute a system command without leaving BASIC. If an operand is provided, that command is executed immediately. If there is no operand, the terminal user is returned to system mode. The user may resume BASIC by issuing the RESUME command.

Format:

```
SYSTEM [system-command]
```

Examples:

```
SYSTEM
SYSTEM FSTATUS JOBS
```

6. BASIC Program Techniques

6.1. INTRODUCTION

Constructing a BASIC program requires translation of the problem into a set of statements that the BASIC system can use in solving the problem. To aid in selecting the proper statements needed to solve a specific problem, a summary of statement and command formats is provided in Appendix A. Once the required statements and commands are selected, refer to the detailed descriptions of those statements and commands in Sections 3 through 5 to review their characteristics and restrictions.

In translating a problem into a series of statements, the user should be familiar with the hierarchy of arithmetic operations, the use of loops, tables, lists, built-in functions, and multiline functions in BASIC. These subjects are covered in detail in this section.

6.2. HIERARCHY OF ARITHMETIC OPERATIONS

BASIC can perform simple operations such as addition, subtraction, multiplication, division, and exponentiation. BASIC can also evaluate numerous built-in functions and user-defined functions. The order in which the simple operations, built-in functions, and user-defined functions are evaluated are similar to those used in standard mathematical calculation, with the exception that all BASIC operations must be written on a single line.

The five simple operators that can be used in BASIC are:

Operator	Definition	Example
**	Exponentiation	A**B
*	Multiplication	A*B
/	Division	A/B
+	Addition	A+B
-	Subtraction	A-B

The hierarchy of arithmetic operations is summarized in the following rules:

1. The arithmetic expression enclosed in parentheses is evaluated first, and its value may then be used in further computations.

   Example:

   ```
   X*(A+B)
   ```

   In this example, the expression A+B is evaluated first and its value is then multiplied by X.

2. Where parentheses are omitted, or where the entire arithmetic expression is enclosed within a single pair of parentheses, the order in which the operations are performed is as follows:

Operation	Hierarchy
Evaluation of functions (built-in or user-defined)	1st (highest)
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th

Example 1:

```
A*B/C** SQR(D)+E
```

This arithmetic expression is evaluated in the following order:

a. SQR(D) Call the result T1 (function)

b. C**T1 Call the result T2 (exponentiation)

c. A*B Call the result T3 (multiplication)

d. T3/T2 Call the result T4 (division)

e. T4+E Final operation (addition)

Also, for operators of the same hierarchy (with the exception of exponentiation), the component operations of the expressions are performed from left to right.

Example 2:

```
A*B/C
```

This arithmetic expression is evaluated in the following order:

a. A*B Call the result T1

b. T1/C Final operation

Example: 3

```
A**B**C
```

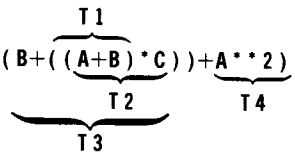
This arithmetic expression is evaluated in the following order:

a. A**B Call the result T1

b. T1**C Final operation

3. Where nested pairs of parentheses are used, the arithmetic expression within the parentheses is evaluated before the outer operations are performed.

Example:



This arithmetic expression is evaluated in the following order:

a. (A+B) Call the result T1

b. (T1*C) Call the result T2

c. B+T2 Call the result T3

d. A**2 Call the result T4

e. (T3+T4) Final operation

6.3. USE OF LOOPS

It is sometimes necessary to construct BASIC programs in such a way that certain portions are performed more than once, with perhaps only slight changes each time. This repeated execution of the same portion of a program is referred to as a loop.

The use of loops can best be illustrated and explained by the following two examples. Both perform the task of printing out a table of the first 100 positive integers together with the square root of each.

Example 1:

```
10          PRINT 1, SQR(1)
20          PRINT 2, SQR(2)
30          PRINT 3, SQR(3)
.
.
.
1000        PRINT 100, SQR(100)
1010        END
```

Without a loop, this example requires 101 statements.

Example 2:

```
10          LET X=1
20          PRINT X, SQR(X)
30          LET X=X+1
40          IF X<=100 THEN 20
50          END
```

With a loop, this example obtains the same table values but with only 5 statements instead of 101. Note that statement number 10 is executed only once, whereas the sequence of statements 20, 30, and 40 are repeated 100 times.

In general, all loops contain four characteristics: initialization (e.g., statement 10), the body (e.g., statement 20), modification (e.g., statement 30), and the exit test (e.g., statement 40).

Because loops are so important and because loops of the type just illustrated arise so often, BASIC provides two statements (FOR and NEXT) to specify a loop.

Example 3:

```
10        FOR X=1 TO 100
20        PRINT X, SQR(X)
30        NEXT X
40        END
```

In this example, the FOR statement initializes the loop index X to 1, the final value to 100, and the step value to 1. Thus, the loop (statements 10 to 30) is performed 100 times and the resulting table is the same as that produced by examples 1 and 2.

Note that the step value can be adjusted by writing the FOR statement in the following form:

```
10        FOR X=1 TO 100 STEP 5
```

In this case, the resulting table contains the integer numbers 1, 6, 11, . . . 96 with their corresponding square roots. Observe that another step of 5 would cause the loop index X to exceed 100.

The STEP value may be positive or negative and may be a decimal number. If statement number 10 in example 3 was changed to the following statement:

```
10        FOR X=100 TO 1 STEP - .1
```

the resulting table would be printed in reverse order and contain the numbers 100, 99.9, 99.8, . . ., 1.1, 1.0 along with their corresponding square roots.

More complicated FOR statements may be written that permit the user to specify the initial, final, and step values as arithmetic expressions. For example, if N and Z have been defined earlier in the program, the user could write the following FOR statement:

Example 4:

```
100        FOR X=Z TO N STEP (N-Z)/10
```

The user should refer to the programming notes of the FOR and NEXT statements in Section 3 for further details about the loop parameters.

Loops within loops may be used and are referred to as *nested loops*. The FOR and NEXT statements may be used for this purpose and are illustrated in Table 6-1. As can be seen in the table, loops may be nested several levels (maximum of 10), but are never permitted to overlap.

Table 6—1. Nested Loops

Allowed	Allowed	Not Allowed
FOR X FOR Y NEXT Y NEXT X	FOR X FOR Y FOR Z NEXT Z FOR W NEXT W NEXT Y FOR Z NEXT Z NEXT X	FOR X FOR Y NEXT X NEXT Y

6.4. USE OF LISTS AND TABLES

In addition to the ordinary variables used in BASIC, there are variables that can be used to designate the elements of a list or a table. These are used where a subscript or a double subscript ordinarily might be used, for example, the coefficients of a polynomial ($a_0,a_1,a_2,\ldots$) or the elements of a matrix. The variables used in BASIC consist of a single letter, which is called the name of the list, followed by the subscripts in parentheses. Thus, the user might write A(0), A(1), A(2), etc, for the coefficients of the polynomial and B(1,1), B(1,2), etc, for the elements of the matrix.

The user can enter the list A(0),A(1), . . . A(10) into a program very simply by the following statements:

Example 1:

```
10         FOR T=0 TO 10
20         READ A(T)
30         NEXT T
40         DATA 2, 3, -5, 7, 2.2, 4, -9, 123, 4, -4, 3
```

Lists and tables with more than 10 subscripts require a DIM statement to indicate that more main storage is needed. For example, a list of 15 numbers may be entered as follows:

Example 2:

```
10        DIM A (25)
20        READ N
30        FOR T=1 TO N
40        READ A(T)
50        NEXT T
60        DATA 15
70        DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
80        END
```

In this example, statements 20 and 60 could have been eliminated and statement 30 replaced by 30 FOR T=1 TO 15. However, this program as typed allows for the lengthening of the list simply by changing statement 60, as long as the value read in for N does not exceed 25.

A simpler way of performing the same function as lines 30 to 50 is to use a MAT statement:

```
30      MAT      READ      A      (N)
```

Matrix A will be redimensioned to the current value of N; a value will then be assigned to each element of A from 1 to N.

A table consisting of three rows and five columns could be entered into a program by writing the following statements:

Example 3:

```
10        FOR T=1 TO 3
20        FOR J=1 TO 5
30        READ B(T,J)
40        NEXT J
50        NEXT T
60        DATA 2, 3, -5, -9, 2
70        DATA 4, -7, 3, 4, -2
80        DATA 3,-3,5,7,8
.
.
.
```

Here again, the user may enter a table with no dimension statement, and the system will handle all the entries from B(0,0) to B(10,10). If a table with a subscript greater than 10 is entered without a DIM statement, an error message specifying a subscript error is generated. This is easily rectified by entering the following statement (assume a 20 by 30 table is required):

```
5         DIM B(20,30)
```

Here, again, a single statement can replace lines 10 to 50:

```
10      MAT      READ      B(3,5)
```

The single letter denoting a list or a table name may also be used to denote a simple variable without confusion. However, the same letter may not be used to denote both a list and a table in the same program. The form of the subscript is flexible. The user might have the list item B(I+K) or the table items B(L,K) or Q(A(S,7),B-C).

6.5. USE OF BUILT-IN FUNCTIONS

The built-in functions provided in BASIC consist of:

- mathematical functions (SIN, COS, TAN, COT, ATN, EXP, LOG, ABS, and SQR);
- specialized functions (INT, RND, SGN, TIM, DET, LEN, MOD, POS, VAL, and EBC);
- string functions (CHR$, CLK$, DAT$, SEG$, STR$, and USR$); and
- file functions (LOC, LOF, MAR, PER, TYP, and NUM).

Examples of each function are provided.

6.5.1. Mathematical Functions

- SIN(x), COS(x), TAN(x), COT(x), and ATN(x) designate the functions sine, cosine, tangent, cotangent, and arctangent, respectively, and the argument x is an angle measured in radians.

  Example:

```
10      X=3.14159/2
20      Y1=SIN(X)
30      Y2=COS(X/2)
40      Y3=TAN(X/3)
50      Y4=COT(X/6)
60      PRINT X,Y1,Y2,Y3,Y4,
70      END
```

  In this example, X is $\pi/2$ (90 degrees), Y1 is the sine of 90 degrees, Y2 is the cosine of 45 degrees, Y3 is the tangent of 30 degrees, and Y4 is the arctangent of 15 degrees.

- EXP(x) designates exponentiation e^x

```
10      E=EXP (X**2)
```

  In this example, E is e^{x2}

- LOG(x) designates the natural logarithm of x, ln(x).

```
10          A=LOG(Y**10)
```

In this example, A is 10 ln(Y).

- ABS(x) designates the absolute value of x, $|x|$.

```
10          B=ABS(-X*Y)
```

In this example, B is $|-X*Y|$.

- SQR(x) designates the square root of x, $\sqrt{x}$.

```
10          C=SQR(A**2+B**2)
```

In this example, C is $\sqrt{A^2+B^2}$.

6.5.2. Specialized Functions

- INT(x) designates the largest integer not exceeding x.

By definition, the following relationships hold:

- If $x > 0$, then INT (x) $\leqslant$ x
- If $x = 0$, then INT (x) = 0
- If $x < 0$, then INT (x) $\leqslant$ x

```
10          X=INT(2.985)
20          Y=INT(-2.015)
30          Z=INT(X-Y)
```

In this example X is 2, Y is -3, and Z is 5 (i.e., Z=INT(2-(-3))).

The INT function can be used to round to any specific number of decimal places. For example, INT(X*10+.5)/10 will round X correct to one decimal place, INT(X*100+.5)/100 will round X correct to two decimal places, and INT(X*10**D+.5)/10**D will round X to D decimal places.

- RND(x) generates a pseudorandom number as follows:
  - If $X > 0$, then RND(x) is a function of X whose value is in the open interval [0, 1).
  - If $X < 0$, the system supplies an arbitrary random number on the open interval [0, 1).
  - If $X = 0$, the system supplies a pseudorandom number, which is a function of the previous random number generated by RND. If $X = 0$, the first time RND is called in a program, the system will supply a fixed number in the open interval [0, 1).
  - If X is not specified (i.e., RND), then RND(0) is assumed.

To generate a sequence of pseudorandom numbers, the user would call any of these options followed by repeated calls to option c.

```
 5        X=Ø
1Ø        FOR L=1 TO 2Ø
2Ø        PRINT RND(x),
3Ø        NEXT L
4Ø        END
```

- RANDOMIZE may be used to cause RND to supply arbitrary random numbers. It is equivalent to call RND (-1). The execution of the previous program would cause the following 20 random numbers to be output:

```
1Ø        RANDOMIZE
```

```
Col 1          Col 16         Col 31         Col 46         Col 61
.763242E-Ø5    .25Ø198        .753869        .567Ø54        .589ØØ2
.747568        .44Ø211E-Ø1    .554667E-Ø1    .252568        .442911
.816485E-Ø1    .52Ø82         .99271         .Ø41932        .572162
.397Ø55E-Ø1    .58698         .8Ø1253        .882914        .793956
```

If the user wants 20 random 1-digit integers, statement 20 is changed as follows:

```
2Ø        PRINT INT(24*RND(x)+5);
```

This results in the following output:

```
Col 1                                                    Col 78
Ø  2  7  5  5  7  Ø  Ø  2  4  Ø  5  9  Ø  5  Ø  5  8  8  7
```

The user can vary the type of random numbers desired. If the user wants 20 random numbers ranging from 5 to 24 inclusive, statement 20 is changed to the following:

```
2Ø        PRINT INT(24*RND(x)+5);
```

In general, if random numbers are chosen within the range A≤RND(x)<A+B, the random function is used as follows:

```
INT (B*RND(x)+A)
```

- SGN(x) designates the sign of x.

$$SGN(x) = \begin{cases} +1, & \text{if } x > Ø \\ Ø, & \text{if } x = Ø \\ -1, & \text{if } x < Ø \end{cases}$$

```
1Ø        X=SGN(Ø)
2Ø        X1=SGN(-1.82)
3Ø        X2=SGN(X1)
4Ø        X3=SGN(-X1)
```

The following example assigns 0 to X, -1 to X1, -1 to X2, and +1 to X3.

- DET designates the value of the determinant of the last matrix to be inverted, or a value of zero if it could not be inverted.

```
10    DIM A(3,3), B(3,3)
20    MAT READ A
30    MAT B = INV (A)
40    MAT PRINT B
50    ''THE VALUE OF ITS DETERMINANT IS'';DET
```

- LEN (X$) returns the length of the string argument.

```
10    LET A$=''ABC''
20    LET B$=A$&A$
30    PRINT LEN (A$), LEN (B$), A$, B$
40    END
```

would print out

```
3       6       ABC       ABCABC
```

- MOD (X,Y) computes the modulus remainder

```
590   MOD (X,Y) = X - Y (INT (X/Y))
600   FOR I=1 TO 5
610   PRINT I; ''MODULO 2 EQUALS'';MOD(I,2)
620   NEXT I
999   END
```

This program would print

```
1  MODULO   2   EQUALS   1
2  MODULO   2   EQUALS   0
3  MODULO   2   EQUALS   1
4  MODULO   2   EQUALS   0
5  MODULO   2   EQUALS   1
```

- POS (A$, B$, X) begins searching A$ at X for the string B$ and returns the position of B$ in A$.

```
10    LET X$= ''THIS STRING IS A TEST''
20    PRINT ''ENTER BEGIN, STRING:'';
30    INPUT Q,Q$
40    PRINT POS(X$,Q$,Q)
50    GOTO 20
60    END
```

If run, this would result in:

```
ENTER BEGIN, STRING;?    1,IS
3
ENTER BEGIN, STRING;?    5,IS
13
ENTER BEGIN, STRING;?    4,DUMMY
0
ENTER BEGIN, STRING;?    STOP
```

- TIM returns the elapsed running time in seconds, accurate to milliseconds.

```
10     LET A=TIM
20     FOR     I=1 to 1000
30     NEXT I
40     PRINT ''ELAPSED TIME IS'', TIM-A
50     END
```

This would print

```
ELAPSED TIME IS     .903999
```

- VAL(Q$) returns the value of the number whose decimal representation is in Q$.

```
10     LET F9$=''4334.57''
20     PRINT VAL(F9$), VAL(SEG$(F9$,3,5))
99     END
```

This program would print

```
4334.67     34
```

In this example, the SEG$ function creates a substring of characters 3, 4, and 5 (which are 34.), and performs the VAL function on this substring.

- EBC (string) obtains the EBCDIC value for a single EBCDIC symbol. Certain symbols cannot be typed and must be entered as 2 or 3 character mnemonics. To interpret a lowercase letter, you must prefix the letter in question with *LC* (e.g., LCE interprets lowercase e). Table 2-1 lists these mnemonics, along with the decimal value which the EBC function will return. EBC is a compile-time, rather than a run-time, function. Examples of using the function follow:

```
EBC(1)=241      EBC(CR)=13
EBC(B)=194      EBC(NUL)=0
```

6.5.3. String Functions

- CHR$(x) returns a 1-character string consisting of the EBCDIC character with the code MOD(INT(x),256). This function may be used to embed special characters or control sequences in printed output:

```
10    PRINT ''THIS SENTENCE IS UNDERLINED'';
20    PRINT CHR$(13);
30    PRINT ''_____''
99    END
THIS SENTENCE IS UNDERLINED
```

Line 20 uses the decimal value of a carriage return, 13, to move the teletype print head back to the start of the output line without skipping down one line (no line-feed is used). This could have also been done by:

```
20    PRINT CHR$ (EBC(CR));
```

- CLK$ gives the time of day in string format.

An 8-character string in the form "hh:mm:ss" is returned.

```
10    PRINT ''THIS PROGRAM WAS RUN AT:''CLK$
20    PRINT
.
.
.
99    END
```

If executed, this program would begin by printing

```
THIS PROGRAM WAS RUN AT: 14:05:30
```

- DAT$ may be used to obtain the current date as an 8-character string in the form yy/mm/dd.

```
10    PRINT ''THIS PROGRAM WAS RUN AT'';CLK$;''ON'';DAT$
20    PRINT . . .
.
.
.
99    END
```

This program would begin by printing

```
THIS PROGRAM WAS RUN AT 14:06:10 ON 80/06/24
```

- SEG$(A$,X,Y) allows the user to obtain substrings of a larger string. All characters between positions X and Y inclusive of A$ will be returned as a new string. If X>Y, then a null string is returned. The appropriate beginning or end of A$ is returned in the case where X <=0 or Y>LEN(A$).

  1. If CLK$ is 14:10:05, then SEG$(CLK$,1,5) would be 14:10.

  2. The function call SEG$(B1$,2,4095) would always return a string consisting of all but the first character of B1$.

  3. The function call SEG$ (C$,1,LEN(C$)-1) would always return a string consisting of all but the last character of C$.

- STR$ (x) may be used to convert a floating point number to its decimal representation. This function returns a string.

```
10    LET N2=6.35
20    PRINT  STR$(N2),   SEG$(STR$(N2),2,2)
```

  This would print

```
6.35  6
```

  Notice that STR$(VAL(A$))=A$ and VAL(STR$(X)) = X. STR$ and VAL are inverse functions.

- USR$ designates the logon-id of the user who is currently executing the program. This is a 6-character string derived from the user-id stated on the LOGON command. ←

6.5.4. File Functions

- LOC (#n) returns the current location of the file pointer for the file assigned to channel n. This function is useful if a program must remember the location in the file to be referenced later.

```
10 FILE #3: ''PROG,DISKFILE,PACK37    ''
     .
     . (processing)
     .
20 READ #3: A6$
21 LET R=LOC(#3)-1
     .
     . (process record in A6$)
     .
30 RESET #3: R
31 WRITE #3: A6$
```

  In this example, the current location pointer is in some unknown position when statement 20 is executed, but the record at that position must be read, changed, and written back. Statement 21 obtains the current position and decrements it because the READ statement automatically increments the location pointer. The record can then be processed. To overwrite a record, the file is reset back to the record by statement 30 and written by statement 31.

- LOF (#n) returns the current value of the end-of-file pointer for the file assigned to channel n. This value is equivalent to the number of records in the file.

```
170 FILE #2:'','ERRORS',SYSRES    ''
180 FOR I=1 TO LOF(#2)
190 WRITE #2:A$(I)
200 NEXT I
```

In this example, the value of LOF is used to control a FOR loop. Each record in the file is written from the corresponding array element in A. This same function can be accomplished with the file IF statement:

```
170 FILE #2: '',ERRORS',SYSRES    ''
180 I=1
190 IF END #2 THEN 230
200 WRITE #2:A(I)
210 I=I+1
220 GOTO 190
230 . . .
```

- PER (#n,A$) allows the user to determine if a file operation will be permitted if executed against the specified file. The function specified by string expression A$ is tested against the file assigned to channel n. a +1 is returned if the function is permitted, 0 if not, and -1 if an invalid function statement is used.

```
210 PRINT ''ENTER NAME OF FILE TO PROCESS:'';
220 INPUT N1$
230 FILE #3:N1$
240 IF PER(#3,''INPUT'')=1 GOTO 300
250 PRINT ''FILE CAN'T BE READ, ENTER CORRECT FILE WITH PASSWORD''
260 GOTO 210
300 PRINT ''FILE NAME ACCEPTED''
.
. continue processing
.
```

This would result in:

```
ENTER NAME OF FILE TO PROCESS:? sq,myfile,mypack
FILE CAN'T BE READ, ENTER CORRECT FILE WITH PASSWORD
ENTER NAME OF FILE TO PROCESS:? sq,myfile(pass),mypack
FILE NAME ACCEPTED
```

In this example, the user must enter a file for the program to process. The program will later read the file using INPUT statements. To avoid program termination should BASIC not permit this, the PER function tests if INPUT is accepted for the file. The most likely reason for it not being accepted is the failure to enter the correct READ password.

- TYP (#n,A$) allows the user to test the file type of a file. The string expression A$ specifies one of the possible file types to test against the file at channel n; a +1 is returned if the file has that type, 0 if not, and -1 if an illegal file was specified by A$.

```
300 IF TYP (#3,''LIBRARY'') = 1 GOTO 330
310 PRINT ''SPECIFY ONLY LIBRARY FILES WITH THIS PROGRAM''
320 GOTO 210
330 PRINT ''FILE ACCEPTED''
```

This example is a continuation of the last example and shows how a program that is designed to run using only library files can test user-supplied files.

- NUM can be used with MAT INPUT of vectors to determine how many elements of the vector were entered.

```
*710 DIM V(100)
*720 PRINT ''ENTER LIST OF NUMBERS''
*730 MAT INPUT V
*740 S=0
*750 FOR I = 1 TO NUM
*760 S=S+V(I)
*770 NEXT I
*780 PRINT ''SUM OF NUMBERS IS:'';S;  ''AVERAGE IS:'';S/NUM
*790 END
*RUN
ENTER LIST OF NUMBERS
?1,9,8,2,3,4,8,&
?45,20,16
SUM OF NUMBERS IS: 116, AVERAGE IS 11.6
*
```

In this example, a vector is used to accept a variable number of input values from the terminal. The NUM function is then used to determine how many elements of the vector are to be processed. An ampersand (&) was used on the first line of input from the terminal because the entire list would not fit on one line.

6.6. USE OF MULTILINE FUNCTIONS

Multiline functions are defined using a combination of DEF and FNEND statements. The user should refer to the programming notes on the DEF and FNEND statements in Section 3 for further details concerning the construction of multiline functions.

Example:

```
110        DEF FNA(N)T,H
120        REM THIS MULTILINE FUNCTION COMPUTES
130        REM THE FACTORIAL OF N
140        T=1
150        IF N<=1 GOTO 190
160        FOR H=2 TO N
170        T=T*H
180        NEXT H
190        FNA=T
200        FNEND
```

If this multiline function is called within the following sequence of statements:

```
10          FOR J=0 TO 9
20          PRINT J;''! △='';FNA(J)
30          NEXT J
40          END
```

the printed output appears as follows:

```
Col 1
↓
0! △=△1
1! △=△1
2! △=△2
3! △=△6
4! △=△24
5! △=△120
6! △=△720
7! △=△5040
8! △=△40320
9! △=△362880
```

6.7. USE OF SUBPROGRAMS

Subprograms provide a mechanism by which independent, parameterized routines can be developed and called with minimal program overhead.

The following example shows a simple subprogram that translates strings, which may contain lowercase characters, to all uppercase. The calling program need only issue a CALL statement selecting the subprogram and stating which string is to be converted. Upon return from the routine, the string will contain only uppercase characters. Although this main program converts a file from uppercase/lowercase text to all uppercase, other programs could use the subprogram for other purposes if it were saved in a common library.

Example:

```
100 FILE #4: ''TEXT,LIBFILE(RDPASS)''
110 FOR I=1 TO LOF (#4)
120 LINPUT #4:L$
130 CALL ''UPPER'':L$
140 RESET #4:LOC(#4)-1
150 WRITE #4:L$
160 NEXT I
170 END
500 SUB ''UPPER'':S$
510 DIM C(128)
520 CHANGE S$ TO C
530 FOR I=1 TO C(0)
540 IF C(I)>EBC(Z)-64 GOTO 600
550 IF C(I)<EBC(A)-64 GOTO 600
560 C(I)=C(I)+64
600 NEXT I
610 CHANGE C TO S$
620 SUBEND
```

This example also makes use of the CHANGE statement to separate each character of the string and convert each to its EBCDIC value. Each character value can then be tested for lowercase and, if true, changed to uppercase by adding decimal 64, which is the decimal difference between the EBCDIC characters A and a. After the individual characters have been processed, they are combined into a string via the CHANGE function.

6.8. USE OF FILES

Several examples of programs using files are presented in this subsection.

The following BASIC program uses several files to operate on library elements. The purpose of this program is to read a COBOL program, locate any references to the COBOL 'COPY' verb, and insert the copied modules inline.

Example 1:

```
100 PRINT ''ENTER COBOL PROGRAM NAME AND COPYLIB FILE NAME'';
200 INPUT P$,C$
300 FILE #1:P$
400 RENAME #1:''*''
500 LINPUT #1:R1$
600 IF POS (R1$,''IDENTIFICATION DIVISION'',1) = 0 THEN 8000
700 RESET #1
800 FILE #2:''*''
```

This portion of the program queries the terminal user for the COBOL program name and the name of the file where the copy elements can be found. The file is opened and immediately renamed to temporary file to prevent overwriting the original module on errors. The first record is then read and tested to see if it is a valid COBOL program. If not, the user is notified. Otherwise, the file is reset so it can be reread from the beginning.

Example 2:

```
1000 FOR I = 1 TO LOF (#1)
1100 LINPUT #1:R1$
1200 LET C = POS (R1$, ''COPY'', 7) + 1
1300 IF C-1>0 THEN 3000
1400 WRITE #2: R1$
1500 NEXT I
1600 RENAME #2: P$
1700 GOTO 9999
```

The program file is now read, one line at a time, and tested for the COPY verb. If the record is other than a COPY, it is written to the output file. Otherwise, a separate section of code is used to process the copy. Finally, the output file is renamed to the original file name so that when it is closed it will be written in place of the original.

Example 3:

```
3000 CALL ''FINDNOSP'': R1$, C+4, C2
3100 LET C+ = POS (R1$&'' '','' '',C2)
3200 IF SEG$ (R1$, C3-1, C3-1) <> ''.'' THEN 3400
3300 LET C3=C3-1
3400 LET N$ = SEG$ (R1$, C2, C3-1)
3500 FILE #3: N$& '','' &C$
3600 RENAME #3: ''*''
3700 FOR J =1 TO LOF (#3)
3800 LINPUT #3: R2$
3900 WRITE #2: R2$
4000 NEXT J
4100 GOTO 1500
```

Once a COPY statement has been found, the copied module name must be isolated. This is concatenated onto the file name and the library element is opened. It too is renamed to a temporary file so it is not overwritten. Each statement of the element is then added to the output file.

Example 4:

```
8000 PRINT ''THIS IS NOT A COBOL PROGRAM, TRY AGAIN''
8100 GOTO 100
9999 END
```

These statements complete the main program.

The subprogram FINDNOSP must also be written. Its purpose is to find the first nonblank character in a string. It is called with three parameters, a string to search, the column beginning the search, and a variable into which the result is placed. The subprogram scans the string and returns the column of the first nonblank character in the string after the column specified by parameter 2; the result is returned in parameter 3. If nonblanks are not found, zero is returned.

Example 5:

```
10000 SUB ''FINDNOSP'': S$, B, E
10100 FOR E = B TO LEN (S$)
10200 IF SEG$ (S$, E, E) <> '' '' GOTO 19999
10300 NEXT E
10400 E = 0
19999 SUBEND
```

6.9. HINTS FOR MORE EFFICIENT CODE

The following suggestions for writing BASIC programs improve the execution time and reduce main storage requirements:

- Use intrinsic system functions instead of BASIC code whenever possible.
- Use FOR loops rather than maintaining counters in BASIC.
- Use string functions, such as POS and SEG$, rather than maintaining an array of characters stored one character per word.
- Use MAT statements to process matrixes, rather than indexing with FOR loops.
- Rather than using several LET statements to compute a result, combine them into a single LET statement. This avoids saving temporary values and is especially helpful for string manipulation.
- When using DATA statements, combine several values onto one statement rather than using one value per statement. The result is a faster RUN compilation.
- Use the RCSZ parameter on file parameter strings rather than using the MARGIN statement. This results in a much faster execution time. ←

7. Errors and Debugging

7.1. GENERAL

There are two basic categories of error:

1. those that prevent the running of the program; and

2. those that permit the program run, but cause wrong answers or no answers to be printed. (These latter errors are called logic errors).

7.2. ERRORS PREVENTING RUNNING OF PROGRAM

It may occasionally happen that the first run of a new program is free of errors and gives the correct answers. But it is much more common that errors are present and have to be corrected. Errors that prevent the running of the program are detected by the syntax checker, the editing command processor, the system monitor processor, the run-time error routines, and the post-compilation routines. (The errors reported by all of the system components mentioned, except the syntax checker, are listed in Appendix C. This appendix also suggests, for each error, the procedure to correct the error condition.)

The syntax checker detects improper syntax in each statement and reports the error by printing a question mark (?) on the terminal followed by a copy of the incorrect statement, up to but not including the first character in error.

Example:

```
10 FOR N=1, 7
```

Because the comma is not permitted in a FOR statement, the system responds with the following message:

```
? 10 FOR N=1
```

and waits for the user to complete the statement.

The user types in the following response:

```
TO 7
```

Now, the following corrected, complete statement is successfully processed by the system:

```
10 FOR N=1 TO 7
```

7.3. LOGIC ERRORS

Logic errors are those that permit the program to run but cause wrong answers or no answers to be printed. In either case, after the errors are discovered, they can be corrected by changing, inserting, or deleting statements from the program. A statement is changed by typing it correctly with the same line number. A statement is inserted by typing it with the new line number. A statement is deleted by either typing the line number and pressing the TRANSMIT key or using the DELETE command.

Corrections to a BASIC program can be made at any time either before or after a run. In addition, line numbers may be typed in out of sequence because BASIC arranges them in ascending order once they are read.

The following program reads in a series of numbers and finds the largest and smallest numbers in the series. The program also computes the average of the series.

Example:

```
10        INPUT N,A
20        L=S=A
30        FOR T=2 TO N
40        INPUT X
50        A=A+X
60        IF X >=L THEN 90
70        L=X
80        GOTO 110
90        IF X >=S THEN 110
100       S=X
110       NEXT T
120       A=A/N
130       PRINT ''SMALL='';S,''LARGE='';L,''AVERAGE='';A
140       END
```

The user types in the following data values when this program is executed:

```
5,1
2
3
4
5
```

The resulting output appears as:

```
SMALL=△1          LARGE=△1          AVERAGE=△3
```

The value for LARGE is obviously incorrect. After examining the program, it becomes evident that the IF statement on line number 60 should be changed as follows:

```
60 IF X<=L THEN 90
```

After this correction is made and the program is reexecuted with the same input data, the resulting output appears as:

```
SMALL=△1          LARGE=△5          AVERAGE=△3
```

8. BASIC in a Batch Environment

8.1. INTRODUCTION

Although primarily used in an interactive environment, BASIC can run effectively as a batch program. The requirements are simple. The user would merely enter on cards exactly what he would key in at the workstation during an interactive session, including everything from LOGON to LOGOFF. The user must store this routine as a file either on the spool file, using the spooler IN command, or on a library file via the librarian, using EDT or the interactive service COPY command. Once stored, the user can run the routine by issuing the ENTER command.

8.2. PROGRAMMING CONSIDERATIONS

The batch mode differs from the interactive mode in the following ways:

- BASIC messages with a reply
- BASIC commands or source statements
- Syntax errors in source statements
- RU command

8.2.1. BASIC Messages with a Reply

BASIC assumes a Y response for the following messages with a reply:

BA064 EXECUTION PAUSED AT LINE xxxx CONTINUE (Y/N)? ▷

IS100 FILE/MODULE ALREADY EXISTS; OK TO WRITE IT? (Y,N) ▷

BA118 SOURCE MODULE NOT SAVED - TERMINATE (Y/N)? ▷

8.2.2. BASIC Commands or Source Statements

BASIC prints all input commands and source statements.

8.2.3. Syntax Errors in Source Statements

When an error is found in a source statement, a syntax message error number is printed on the line following the prompt message.

Example:

```
Input:      20 IF A > 10 THEN GO TO 100

Output:     ?20 IF A > 10 THEN

Output:     ERROR MSG # = BA132
```

8.2.4. RU Command

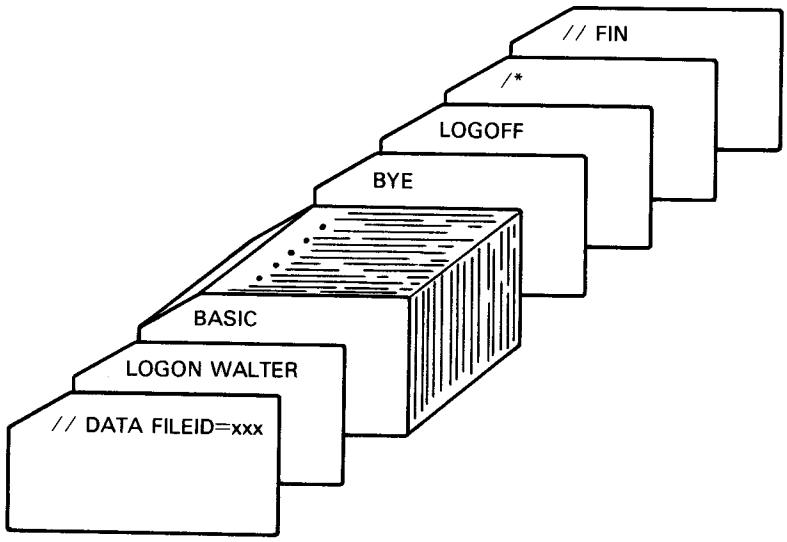
If BASIC finds a syntax error in a source statement, the RU command is ignored. This error condition will be reset by issuing a NEW, an OLD, or RUNOLD command.

Example:

```
Input:      10 A=1

Output:     *

Input:      20 A=A+1

Output:     *

Input:      30 IF A > 10 THEN GO TO 60

Output:     ?30 IF A > 10 THEN

Output:     ERROR MSG # = BA132

Input:      40 PRINT A

Output:     *

Input:      50 GO TO 20

Output:     *

Input:      60 END

Output:     *

Input:      RU

Output:     BA026 UNCORRECTED ERROR IN SOURCE

Output:     *
```

8.3. BASIC BACKGROUND OPERATION

The following is an example of a BASIC background operation:



The card deck of the BASIC batch session is read into the spooler and entered by using the console as follows:

Input: `IN (spooler command)`

Output: `IRØ2 SPOOL FILE BASIC CREATED`

↓

Input: `ENTER Q=RDR,HOLD=N,FILE=xxx`

↓

`stating of BASIC background`

Figure 8-1 illustrates how the system handles BASIC in a batch environment:

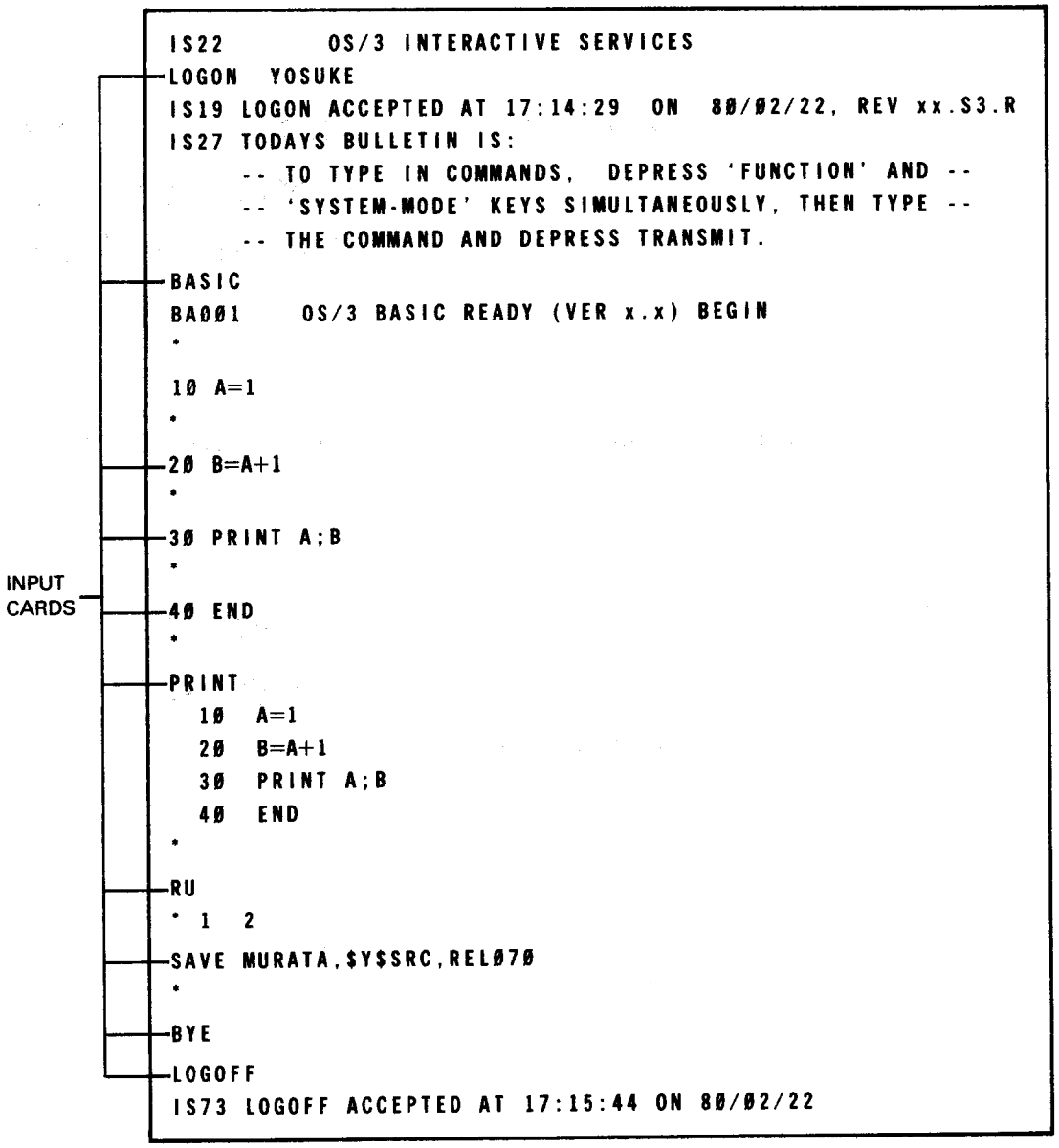


Figure 8—1. BASIC Batch Environment Printout

Appendix A. Summary of BASIC Statement and Command Formats

Table A-1 contains a listing of all of the BASIC statements and commands with examples of each.

Table A—1. BASIC Statement and Command Formats (Part 1 of 5)

Operation	Operand Format	Type	Use	Examples
BYE		Command	Terminates BASIC and returns to SYSTEM.	BYE
CALL	string-constant[:param-list]	Subprogram Statement	Initiates a call to a subprogram.	17 CALL ''SUBR'':3+4,A,B() 18 CALL ''FIND'':#3,SIN, (A) 19 CALL ''SEND'':C(,),K(3,4),B$
CHAIN	{string-expression} {channel-setter} [WITH channel-setter,...]	Subprogram Statement	Initiates compilation and execution of another program segment.	23 CHAIN ''PROGRAM2,CHAINLIB,PACK34'' 24 CHAIN A$ WITH #3 25 CHAIN #4 WITH #1,#4,#J8
CHANGE	{string TO array} {array TO string-variable} [BIT expression]	General Statement	Converts a string to a vector or vice versa.	34 CHANGE A$ TO V 35 CHANGE M TO B3$ 36 CHANGE G TO K1$ BIT 12
DATA	{string-constant}.... {numeric-constant}	Input/Output Statement	Supplies values for subsequent READ statements.	45 DATA 1,3,6,1E3,-.34,17.3E34 47 DATA ''STRING ONE'', STRING TWO, OTHER STR 49 DATA FOURTH STRING, 33, ''FIFTH STRING''
DEF	FNletter[$][(param-list)] [,local-list] [expression]	Declaration	Defines the entry point into a user function.	54 DEF FND (X,Y)=SQR(X**2+Y**2) 55 DEF FNS$(X,Y$)=SEG$(Y$,X,X)&'''' 56 DEF FNQ 57 DEF FNG$,I,J,K 58 DEF FNE (A,B,C),W,Z
DELETE	[line number-list][''search-string'']	Command	Deletes lines from the BASIC program in the workspace.	DELETE 10 DELETE 100-132 DELETE ''INSTRUCTIONS'' DELETE 1-100 ''REM''
DIM	letter[$](integer[,integer])....	Declaration	Defines arrays or vectors and specifies subscript bounds.	67 DIM A(3), B(4,5) 68 DIM G$(45) 69 DIM C(100), H$(2,40)
END		Control Statement	Defines the last statement in the main program and terminates execution.	78 END
FILE	channel-setter:string-expression	Input/Output Statement	Defines and opens a data file.	82 FILE #3:''*'' 83 FILE #1:''SQ,ERRORS,SPOOL3'' 84 FILE #7:''COBOLPGM,LIBFILE(/WRPASS)'' 85 FILE #J:A$
FNEND		Declaration	Defines the end of a multiline user function and returns control.	88 FNEND
FOR	numeric-variable=numeric-expression TO numeric-expression [STEP numeric-expression]	Control Statement	Initiates a loop and specifies values for loop index.	93 FOR I=3 TO 10 94 FOR J2=I TO POS(A$,B$,I) 95 FOR K=J2 TO L3 STEP 4
GOSUB	line-number	Control Statement	Transfers control to a subroutine and saves return address.	102 GOSUB 943
GOTO	line-number	Control Statement	Transfers control to another statement in the program.	111 GOTO 130

Table A—1. BASIC Statement and Command Formats (Part 2 of 5)

Operation	Operand Format	Type	Use	Examples
IF	Format 1: expression test expression {GOTO / GOSUB / THEN} line-number Format 2: {END / MORE} channel-setter {GOTO / GOSUB / THEN} line-number	Control Statement	Compares two expressions according to the "test" specified and, if true, performs the GOTO or GOSUB. A file condition may also be tested.	120 IF A$=''YES'' THEN 340 122 IF SIN(X)=0.5 GOTO 43 123 IF END #3 GOSUB 230
INPUT	[channel-setter:]variable-name,...	Input/Output Statement	Solicits input from the terminal or reads a file and assigns values to the variables listed.	130 INPUT A,B$ 140 INPUT #1: D(3,4),J
LET	Format 1: numeric-variable [=numeric-variable...] =numeric-expression Format 2: string-variable [=string-variable...] =string-expression Format 3: FNletter[$]=expression	Assignment	Assigns values to numeric or string variables or to a function.	143 LET A$=SEG$(A$,3,4) 145 LET B(3,4)=SIN(Y) 147 LET FND=B(3,4)*A(4)+1
LIBRARY	string-constant,...	Subprogram Statement	Specifies names of subprogram libraries to be searched.	155 LIBRARY ''SUBLIBRARY,PACK11'' 157 LIBRARY ''CATALOGEDSUBLIBRARY(ALLOWD)''
LIST	[line-number-list][''search-string'']	Command	Displays lines of a BASIC program to the terminal.	LIST 3-4, 10, 100-200 LIST ''PRINT'' LIST 1-100 ''REM''
MARGIN	[channel-setter:]numeric-expression	Input/Output Statement	Changes the current margin setting for the terminal or a file.	160 MARGIN 120 164 MARGIN #3: 64
MAT	letter=letter+letter	Matrix Operations	Adds two matrixes and places the result in a third matrix.	174 MAT A=B+C 175 MAT V=W+Z
MAT	letter=CON[(trimmer)]	Matrix Operations	Sets all elements of the matrix to the value 1. The matrix may optionally be redimensioned.	178 MAT A=CON 179 MAT V=CON(I)
MAT	letter=IDN[(trimmer)]	Matrix Operations	Sets the matrix to an identity matrix. The matrix may optionally be redimensioned.	185 MAT H=IDN (3,3) 188 MAT J=IDN
MAT	letter=INV(letter)	Matrix Operations	Performs the matrix inversion function on square matrixes.	190 MAT Q=INV(R)
MAT	letter=letter*letter	Matrix Operations	Multiplies two matrixes and places the result in a third.	198 MAT U=V*W 199 MAT A=V*B

Table A—1. BASIC Statement and Command Formats (Part 3 of 5)

Operation	Operand Format	Type	Use	Examples
MAT	letter$=NUL$[(trimmer)]	Matrix Operations	Sets all elements of a string matrix to null strings. The matrix may optionally be redimensioned.	201 MAT D$=NUL$ 205 MAT F$=NUL$(I,J) 206 MAT G$=NUL$(3)
MAT	letter=(numeric-expression)*letter	Matrix Operations	Multiplies all elements of a matrix by a scalar value.	212 MAT D=(J+4)*E 213 MAT V=(SIN(U))*W
MAT	letter=letter-letter	Matrix Operations	Subtracts two matrixes and places the result in a third matrix.	221 MAT D=F-E
MAT	letter=TRN(letter)	Matrix Operations	Transposes rows for columns in a matrix.	234 MAT D=TRN(F)
MAT	letter=ZER[(trimmer)]	Matrix Operations	Sets all elements of the matrix to the value 0. The matrix may optionally be redimensioned.	244 MAT S=ZER 247 MAT E=ZER(3,4)
MAT INPUT	[channel-setter:] letter[$][(trimmer)],...	Matrix Operations	Solicits input from the terminal or a file and assigns values to each element of the matrix.	253 MAT INPUT #3:B$
MAT LINPUT	[channel-setter:] letter$[(trimmer)],...	Matrix Operations	Solicits input from the terminal or a file and assigns complete lines of data to each element of the string matrix.	255 MAT LINPUT #3: A$ 256 MAT LINPUT D$
MAT PRINT	[channel-setter:] letter[$][separator],...	Matrix Operations	Displays a matrix to the terminal or a file. Spacing is determined by the separator.	262 MAT PRINT A, B; C; 265 MAT PRINT #8: B$,
MAT READ	[channel-setter:] letter[$][(trimmer)],...	Matrix Operations	Reads values in for each element of the matrix from DATA statements or from a file.	272 MAT READ A 277 MAT READ B$(3) 279 MAT READ #J+3:D(3,4)
MAT WRITE	channel-setter:letter[$],...	Matrix Operations	Writes each element of the matrix to a record in the file.	281 MAT WRITE #3: A,B 283 MAT WRITE #1:K$,Y
MERGE	element,library[(password)] [,volume]	Command	Reads in an existing program on disk without deleting the original contacts of the workspace.	MERGE SUBR, SUBLIB, SUBPAK
NEXT	numeric-variable	Control Statement	Terminates a loop initiated by a FOR statement.	292 NEXT I 293 NEXT J5
NEW		Command	Deletes the contents of the BASIC workspace so that a new program may be written.	NEW
OLD	element,library[(password)][,volume]	Command	Deletes the contents of the BASIC workspace when located and reads in an old program from disk.	OLD PRINTSIN,PROGRAMLIB,DISKPK OLD COMPUTE,CATALOGUEDFILE
ON	numeric-expression {GOTO / GOSUB / THEN} line-number...	Control Statement	The value of the numeric expression selects which line number in the list will be used with the GOTO or GOSUB statement.	320 ON J*(4+I) GOTO 120,300,120,430 111 ON K GOSUB 10,20,30,50,10,40

Table A—1. BASIC Statement and Command Formats (Part 4 of 5)

Operation	Operand Format	Type	Use	Examples
PAUSE		Control Statement	Suspends execution of the program and queries the terminal user to determine whether to continue or not.	332 PAUSE
PRINT	[line-number-list]['' search-string'']	Command	Displays lines of a BASIC program to the terminal.	PRINT 3-4,10,100-200 PRINT ''LINPUT'' PRINT ''END'' 9000-99999
PRINT	[channel-setter:] expression[separator],...	Input/Output Statement	Displays the value of each expression listed according to the format specified by the separators. Display is to a file or a terminal.	345 PRINT ''THE ANSWER IS'';A3 354 PRINT I,J,K 356 PRINT TAB(I);I;
RANDOMIZE		General Statement	Obtains a random seed for the random number generator.	362 RANDOMIZE
READ	[channel-setter:]variable,...	Input/Output Statement	Assigns values to each of the variables listed from DATA statements or by reading records from a file.	371 READ A,B,C 373 READ #4:A$(45) 377 READ #I: A3, B7$, C(2,3)
REM	any characters for a comment	General Statement	Used for an inline comment.	391 REM THIS PROGRAM COMPUTES THE WVB VALUES 392 REM FOR AN ARRAY 393 REM 394 REMARK
RESEQUENCE	start[:incr][:file-params]	Command	Resequences the program as it is saved to a library file using the starting line number and increment.	RESEQ 100:50:RESPROG, PROGLIB, PACK57
RESET	[channel-setter:[numeric-expression]]	Input/Output Statement	Repositions the file or the DATA statement pointer.	382 RESET 384 RESET #3 388 RESET #I:V3
RETURN		Control Statement	Returns from a subroutine which was called via GOSUB.	395 RETURN
RUN		Command	Initiates compilation of a program in the workspace.	RUN
RUNOLD	element,filename[(password)][,volume]	Command	Initiates compilation of an old program stored on disk.	RUNOLD COMPUTE,CATALOGUEDFILE
SAVE	element,filename[(password)][,volume]	Command	Saves the BASIC program contained in the workspace on disk.	SAVE COMPUTE,CATALOGUEDFILE(PSWORD)
SCRATCH	channel-setter	Input/Output Statement	Deletes the contents of the BASIC file.	403 SCRATCH #3 404 SCRATCH #I-2
STOP		Control Statement	Terminates execution in the program. May be placed anywhere within the program as opposed to END, which must be last.	412 STOP
SUB	string-constant:params	Subprogram Statement	Defines the entry into a subprogram and specifies any passed parameters.	421 SUB ''FINDSPAC''

Table A—1. BASIC Statement and Command Formats (Part 5 of 5)

Operation	Operand Format	Type	Use	Examples
SUBEND		Subprogram Statement	Indicates the last statement in the subprogram and returns control to the CALL statement when executed.	437 SUBEND
SUBEXIT		Subprogram Statement	Returns control to the CALL statement from anywhere within the subprogram.	449 SUBEXIT
SYSTEM	[system command]	Command	Returns control to system, or executes a single system command without leaving BASIC.	SYSTEM SYSTEM FSTATUS PROGRAMLIB,PACK33
SYSTEM	string	Control Statement	Issues a system command from a running BASIC program.	476 SYSTEM''RUN''&P1$
TIME	integer	General Statement	Changes the CPU time limit placed on an executing program.	TIME 120
USING	using-string,expression[,expression],...	Input/Output Statement	Defines format string and edited expressions.	127 PRINT USING A$,B,C 145 MAT PRINT USING ''#,##!!!!!'',B 167 PRINT #7:USING C1$,F$;G
WRITE	channel-setter:expression,...	Input/Output Statement	Writes records to a file, one per expression listed.	523 WRITE #3:A,SIN(X),B$

Appendix B. Sample BASIC Session

An example of a complete session is provided in Figure B-1 to aid the new user when learning BASIC. The designation IN: denotes text, which is supplied by the user, and OUT: designates responses from the system.

```
 1. IN:  LOGON USR1
 2. OUT: IS22 OS/3 INTERACTIVE SERVICE
 3. OUT: IS19 LOGON ACCEPTED AT 13:37:22 ON 80/05/30, REV 7.0S.21
 4. IN:  BASIC
 5. OUT: BA001 OS/3 BASIC READY (VER 7.0) BEGIN
 6. OUT: *
 7. IN:   10 PRINT ''PROGRAM TO COMPUTE AREA OF A CIRCLE GIVEN RADIUS''
 8. OUT: *
 9. IN:   20 PRINT ''ENTER CIRCLE RADIUS:'';
10. OUT: *
11. IN:   30 INPUT R
12. OUT: *
13. IN:   40 A=3.14159 R**2
14. OUT: ?40 A=3.14159
15. IN:   40 A=3.14159 * R**2
16. OUT: *
17. IN:   50 PRINT ''AREA OF A CIRCLE IS'':A,''CONTINUE?'';
18. OUT: *
19. IN:   60 INPUT C$
20. OUT: *
21. IN:   70:IF C$=''YES'' THEN 200
22. OUT: *
23. IN:   80 END
24. OUT: *
25. IN:   RUN
26. OUT: BA024 UNDEFINED LINE 00200
27. OUT: *
28. IN:   LIST 70
29. OUT:  70 IF C$=''YES'' THEN 200
30. OUT: *
31. IN:   70 IF C$=''YES'' THEN 20
```

Figure B—1. Sample BASIC Session (Part 1 of 3)

```
32. OUT: *
33. IN:    RUN
34. OUT: PROGRAM TO COMPUTE CIRCLE GIVEN RADIUS
35. OUT: ENTER CIRCLE RADIUS:▷
36. IN:    1
37. OUT: AREA OF A CIRCLE IS 3.14159     CONTINUE?▷
38. IN:    YES
39. OUT: ENTER CIRCLE RADIUS:▷
40. IN:    2
41. OUT: AREA OF A CIRCLE IS 12.5664     CONTINUE?▷
42. IN:    NO
43. OUT: *
44. IN:    BYE
45. OUT: BA118 SOURCE MODULE NOT SAVED - TERMINATE (Y/N)?▷
46. IN:    Y
47. OUT: BA113 BASIC TASK NORMAL TERMINATION
48. IN:    LOGOFF
49. OUT: IS73 LOGOFF ACCEPTED AT 08:15:00 ON 81/10/23
```

Lines	Description
1–3	These lines constitute the log-on procedure. A user has logged on with a user-id of USR1.
4–6	The BASIC compiler is invoked.
7–12	Program lines 10, 20, and 30 are entered and verified by the syntax checker.
13–14	Line 40 is entered, but is incorrect, so it is rejected by the syntax checker. The statement up to and including the constant 3.14159 is correct, but there is an error after the constant.
15–16	The user corrects the error by inserting a multiplication operator between the constant and the variable R. The line is accepted and verified.
17–24	The rest of the program is entered.
25	A RUN command is entered to execute the program.
26–27	An error is detected by the compiler at line 70. Execution is inhibited and the user's terminal is returned to compilation mode.
28–33	Line 70 is displayed and the reference to line 200 is corrected to use line 20. Execution is again attempted.
34–35	The program begins execution by displaying a heading line and a request for input.
36	Data is supplied for the INPUT statement at line 30.
37–38	The answer is computed and displayed, along with the question, as to whether to continue or not. The user requests continuation.
39–40	The program again requests input and is supplied a value of 2.
41–43	A second answer is computed and displayed. This time the user selects not to continue the program, and so it terminates.

Figure B—1. Sample BASIC Session (Part 2 of 3)

Lines	Description
44	To terminate the BASIC compiler, the BYE command is used.
45–46	The user entered the BYE command without saving this program in the library file. BASIC responds with a BA118 message. The user does not want to save the program and replies with Y.
47–48	A LOGOFF command is issued to end the session.

Figure B—1. Sample BASIC Session (Part 3 of 3)

Appendix C. BASIC Error Messages

BASIC error messages for interactive and batch environments are short and self-explanatory. The error messages are listed here in numerical order. The listing includes possible causes of an error and suggested procedures to follow in response to a message. When used in conjunction with the HELP command messages, these error message explanations can help you locate and correct programming errors. For details on using the HELP command, see 5.2.3.

NOTE:

All error messages for OS/3 BASIC are listed in this appendix. These messages are not included in the system messages manual.

Error Message/HELP Command Message

BA000 **XXX YY BASIC TASK ABNORMAL TERMINATION**
BASIC HAS TERMINATED ABNORMALLY, WHERE XXX IS BASIC'S MOST RECENT ERROR AND YY IS THE DATA MANAGEMENT ERROR CODE (IF ANY). IF THE PROBLEM PERSISTS, SAVE ALL RELEVANT DATA AND CONTACT YOUR SPERRY UNIVAC REPRESENTATIVE.

BA001 **OS/3 BASIC READY (VER XX.XX) BEGIN**
OS/3 BASIC INITIAL MESSAGE

BA002 **BASIC EDITING COMMAND UNRECOGNIZABLE**
EITHER AN INVALID COMMAND HAS BEEN ENTERED OR A BASIC STATEMENT HAS BEEN ENTERED WITHOUT A LINE NUMBER. VALID COMMANDS ARE:

OLD	*NEW*	*SAVE*	*RUN*
PRINT	*HELP*	*BYE*	*DELETE*
LIST	*SYSTEM*	*PRINT*	*RUNOLD*
MERGE	*RESEQUENCE*		

BA003 **INVALID LINE RANGE**
VALID LINE RANGES CONSIST OF SINGLE LINE NUMBERS (A, B) OR RANGES OF LINES (A—B). A LINE NUMBER CONSISTS OF AN INTEGER IN THE RANGE 1—99,999.

BA004 **INVALID SEARCH STRING**
A SEARCH-STRING CONSISTS OF ANY CHARACTER STRING ENCLOSED IN QUOTATION MARKS. IF A QUOTE APPEARS IN THE STRING, IT MUST APPEAR AS "".

BA005 **MISSING FILE PARAMETER**
THE FILE PARAMETER FORMAT IS:
1) MIRAM:"'ELEMENT',FILENAME(PASSWORD),VOL"
2) LIB :"ELEMENT,FILENAME(PASSWORD),VOL"
SEE BASIC PROGRAMMER'S REFERENCE FOR DETAILS.

BA006 **ENTER FILE NAME**
THE USER HAS ENTERED A SAVE, OLD, OR RUNOLD COMMAND WITHOUT SPECIFYING A FILE NAME. SUPPLY THE NAME IN RESPONSE TO THIS MESSAGE.

BA007 **ILLEGAL VAL ARGUMENT**
THE STRING PASSED TO THE VAL FUNCTION DID NOT CONTAIN A VALID NUMBER. THE CONTENTS OF THE STRING MUST BE EITHER AN INTEGER OR A DECIMAL NUMBER IN SCIENTIFIC NOTATION. NO EXTRA CHARACTERS MAY BE SUFFIXED TO THE STRING.

BA008 **LOG OF A NON-POSITIVE NUMBER UNDEFINED**
THE LOG FUNCTION HAS ENCOUNTERED A NON-POSITIVE ARGUMENT. THE LOGARITHM OF THIS NUMBER IS UNDEFINED; EXECUTION IS CANCELLED.

BA009 **SQUARE ROOT OF A NEGATIVE NUMBER UNDEFINED**
THE SQR FUNCTION ENCOUNTERED A NEGATIVE ARGUMENT. THE SQUARE ROOT OF THIS NUMBER IS UNDEFINED; EXECUTION IS CANCELLED.

Error Message/HELP Command Message

BA010 **EXPONENT UNDERFLOW, EXECUTION CONTINUES**
*THE RESULT (OR INTERMEDIATE RESULT) OF A COMPUTATION IS LESS THAN THE SMALLEST NUMBER THE HARDWARE IS CAPABLE OF HANDLING. THE NUMBER IS APPROXIMATELY 10 ** —78. ZERO IS SUPPLIED AND EXECUTION CONTINUES.*

BA011 **EXPONENTIATION ERROR**
*INVALID OPERANDS WERE USED WITH THE A**B OR A↑B FUNCTION. THIS ERROR CAN OCCUR IF "A" IS NEGATIVE AND "B" IS NOT AN INTEGER BETWEEN 1 AND 15 OR —1 AND —15.*

BA012 **MATRIX DIMENSIONS ARE INCORRECT FOR FUNCTION**
THE ROW OR COLUMN DIMENSION OF THE MATRICIES IN THE MATRIX STATEMENT IS INCORRECT. CHECK DIM STATEMENT FOR THE MATRICIES IN QUESTION.

BA013 **SAME MATRIX APPEARS ON BOTH SIDES OF EQUAL SIGN**
THE SAME MATRIX MAY NOT BE REFERENCED ON BOTH SIDES OF AN EQUAL SIGN IN A MAT STATEMENT; A NEW MATRIX MUST BE GENERATED.

BA014 **INVALID TRIMMER IN MATRIX STATEMENT**
EITHER THE TRIMER DID NOT RESULT IN A POSITIVE NUMBER, OR THE RESULTANT ARRAY REQUIRED MORE STORAGE THAN THE ORIGINAL ARRAY.

BA015 **ARRAY SUBSCRIPT OUT OF RANGE**
AN ARRAY SUBSCRIPT, WHICH IS OUT OF THE RANGE SPECIFIED BY THE DIMENSION STATEMENT, HAS BEEN DETECTED. THE SUBSCRIPT IS EITHER LESS THAN ZERO OR GREATER THAN THE NUMBER SPECIFIED AS THE UPPER LIMIT IN THE DIM STATEMENT. IF NO DIM STATEMENT IS USED, THE UPPER LIMIT IS 10.

BA016 **FILE STATEMENT INVALID FOR FILE #0**
THE CHANNEL SETTER SPECIFIED WITH THE FILE STATEMENT RESULTS IN A VALUE OF 0. CHANNEL 0 (THE TERMINAL) CANNOT BE DEFINED BY A FILE STATEMENT.

BA017 **STRING EXCEEDS 4095 CHARACTERS**
A STRING OPERATION HAS PRODUCED A STRING WITH A LENGTH IN EXCESS OF 4095 CHARACTERS. THE MAXIMUM NUMBER OF CHARACTERS PERMITTED IN A STRING IS 4095.

BA018 **CHANGE ERROR**
THE CHANGE OPERATION SPECIFIED BY THE FLAGGED STATEMENT IS NOT VALID. POSSIBLE CAUSES OF THIS ERROR ARE AN INVALID VECTOR OR VECTOR SIZE, INVALID BIT EXPRESSION, INVALID STRING RESULT, OR INVALID VALUE ENCOUNTERED DURING CONVERSION.

BA019 **GIVEN LINE EXCEEDS 80 CHARS WHEN RESEQUENCED**
THE LINE SHOWN, WHEN RESEQUENCED, IS LARGER THAN 80 CHARACTERS. THIS IS AN INFORMATIONAL MESSAGE IN THAT THE COMPLETE RESEQUENCED LINE IS WRITTEN OUT (AND CAN BE MODIFIED BY EDT), BUT IF THE PROGRAM IS LATER READ IN BY BASIC, IT WILL BE FLAGGED WITH AN ERROR FOR BEING OVER 80 CHARACTERS LONG.

BA020 **START AND INCREMENT WILL EXCEED 99999**
THE STARTING NUMBER AND INCREMENT USED IN A RESEQUENCE COMMAND CANNOT BE USED AS THEY ARE BECAUSE THEY WOULD CAUSE ONE OF THE NEW LINE NUMBERS TO EXCEED THE MAXIMUM LINE NUMBER IN OS/3 BASIC. USE A DIFFERENT START OR INCREMENT AND REISSUE THE COMMAND.

BA021 **ERROR IN SOURCE — RESEQUENCE TERMINATED**
ONE OR MORE OF THE SOURCE STATEMENTS READ IN BY AN OLD COMMAND WITH ERRORS HAVE NOT BEEN CORRECTED. ONLY VALID PROGRAMS IN THE WORKSPACE MAY BE RE-SEQUENCED. THIS ERROR INDICATES THAT AT LEAST ONE LINE IS SYNTACTICALLY INCORRECT.

BA022 **GOTO OR GOSUB TO UNDEFINED LINE NUMBER**
A GOTO, GOSUB, OR THEN STATEMENT HAS REFERENCED A LINE NUMBER WHICH DOES NOT APPEAR IN THE PROGRAM.

BA023 **REFERENCED SUBROUTINES NOT FOUND IN LIBRARIES**
ALL USER SPECIFIED LIBRARIES HAVE BEEN SEARCHED, BUT THE SUBPROGRAM LISTED IN THE ERROR MESSAGE COULD NOT BE FOUND.

BA024 **UNDEFINED LINE XXXXX**
THE LINE NUMBER REFERENCED IN A GOTO, GOSUB, ON, OR IF-THEN STATEMENT IS NOT PRESENT IN THE PROGRAM OR FUNCTION. INSERT THE REQUIRED LINE OR REMOVE THE REFERENCE TO IT.

BA025 **END STATEMENT IS MISSING OR MISPLACED**
ALL BASIC PROGRAMS MUST HAVE AN END STATEMENT AS THE LAST LINE. INSERT AN END STATEMENT AND RERUN.

BA026 **UNCORRECTED ERROR IN SOURCE**
ONE OF THE STATEMENTS FLAGGED DURING THE PREVIOUS OLD COMMAND HAS NOT BEEN ELIMINATED OR CORRECTED. THE NUMBER OF THAT LINE IS SHOWN.

BA027 **LOADER ERROR AT LINE XXXXX (IN AAAAAAAA)**
WHEN THE ERROR WAS DETECTED, THE BASIC COMPILER WAS AT THE LINE NUMBER GIVEN BY XXXXX. THIS MESSAGE IS DISPLAYED IN CONJUNCTION WITH ANOTHER ERROR MESSAGE.

BA028 **SECOND DEFINITION OF AN ARRAY NOT ALLOWED**
TWO DIMENSION STATEMENTS HAVE BEEN USED TO DEFINE THE SAME VARIABLE. REMOVE ONE OF THE STATEMENTS AND RERUN.

BA029 **NUMBER OF SUBSCRIPTS FOR ARRAY INCORRECT**
THE VARIABLE THAT HAS CAUSED THE ERROR HAS BEEN DIMENSIONED WITH A DIFFERENT NUMBER OF SUBSCRIPTS THAN WERE FOUND IN THE REFERENCE TO IT.

BA030 **BASIC SOURCE LINES OUT OF ORDER**
THE LINES OF SOURCE IN A BASIC PROGRAM READ IN BY A RUNOLD OR CHAIN STATEMENT ARE NOT IN ORDER. THIS IS MANDATORY. AN OLD COMMAND WILL BRING THIS PROGRAM INTO THE WORKSPACE FOR CORRECTING, AND A SAVE COMMAND WILL SAVE THE PROGRAM.

Error Message/HELP Command Message

BA031 **GOTO INTO OR OUT OF FUNCTION DEFINITION**
A FUNCTION MAY NOT REFERENCE PROGRAM LINES THAT DO NOT OCCUR WITHIN THE BODY OF THE FUNCTION, NOR MAY STATEMENTS OUTSIDE THE FUNCTION DEFINITION REFERENCE STATEMENTS WITHIN THE FUNCTION BODY. THIS APPLIES TO GOTO, GOSUB, ON, AND IF STATEMENTS.

BA032 **FUNCTION DEFINITION WITHIN A FUNCTION**
BASIC HAS DETECTED A FUNCTION WITHIN THE BODY OF ANOTHER FUNCTION DEFINITION. CHECK FOR A MISSING FNEND STATEMENT OR RESTRUCTURE THE FUNCTION.

BA033 **SECOND DEFINITION OF THE SAME FUNCTION**
THE SAME FUNCTION HAS BEEN DEFINED TWICE WITHIN THE PROGRAM. REMOVE ONE DEFINITION AND CORRECT THE PROGRAM. RERUN.

BA034 **NUMBER OF PARAMS IN FUNCTION CALL INVALID**
A MAXIMUM OF 16 PASSED PARAMETERS AND LOCAL VARIABLES MAY BE SPECIFIED ON A FUNCTION DEFINITION LINE. REDUCE THE NUMBER AND RERUN.

BA035 **DEF MUST PRECEDE REFERENCE IF LOCALS ARE USED**
WHEN LOCAL VARIABLES ARE USED IN A MULTILINE USER FUNCTION, THE DEFINITION MUST OCCUR AT A LOWER NUMBERED LINE THAN THE FIRST REFERENCE TO THAT FUNCTION. MOVE THE FUNCTION DEFINITION AND RERUN.

BA036 **SIMPLE VARIABLE INCONSISTENT WITH CALL**
THE CALL AND SUB LINES DIFFER IN THE SPECIFICATION OF A SIMPLE VARIABLE TO BE PASSED TO THE SUBPROGRAM. REMOVE THE INCONSISTENCY AND RERUN.

BA037 **FUNCTION ASSIGNMENT DOES NOT MATCH FUNCTION NAME**
THE NAME OF THE FUNCTION BEING ASSIGNED DIFFERS FROM THE NAME OF THE FUNCTION FROM WHICH IT APPEARS. ONLY THE FUNCTION BEING DEFINED MAY BE ASSIGNED A VALUE.

BA038 **"FNEND" FOUND WITHOUT FUNCTION DEFINITION**
THE FNEND STATEMENT WAS DETECTED, BUT IT WAS NOT AT THE END OF A FUNCTION. REMOVE THE STATEMENT OR PUT IT IN THE CORRECT PLACE AND RERUN.

BA039 **INCORRECT NESTING OF FOR-NEXT STATEMENT**
A FOR OR NEXT STATEMENT WHICH WAS NOT NESTED CORRECTLY WAS DETECTED. POSSIBLE CAUSES ARE:
1) A FOR STATEMENT WITH THE SAME INDEX AS THE PREVIOUS FOR IN THE NEST.
2) A NEXT THAT DOES NOT HAVE THE SAME INDEX AS THE FOR IMMEDIATELY PRECEDING IT.
3) A NEXT STATEMENT THAT DOES NOT FOLLOW ANY PROPER FOR STATEMENT.

BA040 **FUNCTION ASSIGNMENT MUST APPEAR WITHIN FUNCTION**
A VALUE MUST BE ASSIGNED TO A MULTILINE FUNCTION BEFORE THE FNEND STATEMENT. THE FUNCTION VALUE MAY NOT BE DEFINED OUTSIDE THE BODY OF THE FUNCTION.

BA041 **PARAMETER TYPE MIS-MATCH**
THE TYPE OF PARAMETER PASSED TO A FUNCTION/SUBPROGRAM CONFLICTS WITH THE TYPE DEFINED FOR THE FUNCTION/SUBPROGRAM. FOR EXAMPLE, A NUMERIC VARIABLE WAS PASSED WHEN A STRING VARIABLE WAS EXPECTED. COMPARE THE LINE IN ERROR AND THE DEFINITION; CORRECT THE DISCREPANCY.

BA042 **DIMENSIONS INCONSISTENT IN SUB CALL**
THE TYPE OF VARIABLES USED IN THE SUB AND CALL LINES DIFFER. EITHER A SCALAR WAS USED WHERE AN ARRAY WAS EXPECTED OR THE NUMBER OF SUBSCRIPTS IN THE SUB AND CALL LINES DIFFER.

BA043 **# OF FUNCTION PARAMS INCONSISTENT IN CALL**
THE NUMBER OF PARAMETERS PASSED TO A SUBPROGRAM DOES NOT AGREE WITH THE NUMBER STATED ON THE SUB LINE, OR DOES NOT AGREE WITH ANOTHER CALL TO THE SAME PROGRAM.

BA044 **FUNCTION DEF MUST PRECEDE USE IN "CALL"**
IN ORDER FOR A USER FUNCTION TO BE PASSED TO A SUBPROGRAM, IT MUST BE DEFINED. MOVE THE DEFINITION INTO LOWER NUMBERED LINES BEFORE THE CALL.

BA045 **SUBROUTINE CALLING ITSELF**
A CALL STATEMENT HAS BEEN FOUND WHICH REFERENCES THE SUBPROGRAM IN WHICH IT RESIDES. RECURSIVE CALLS OF ANY KIND ARE PROHIBITED.

BA046 **NUMBER OF ARGUMENTS INCONSISTENT**
THE NUMBER AND TYPE OF ARGUMENTS PASSED IN THE CALL STATEMENT DO NOT AGREE WITH THE NUMBER AND TYPE ON THE SUB LINE.

BA047 **SUB NAME IS GREATER THAN 8 CHARACTERS**
THE NAME USED ON A SUB OR CALL STATEMENT MUST BE A STRING CONSTANT WHICH IS NOT LONGER THAN 8 CHARS. CORRECT THE SPELLING OR SHORTEN ITS LENGTH.

BA048 **SUBROUTINE LIMIT OF 30 EXCEEDED**
BASIC WILL NOT ACCEPT MORE THAN 30 SUBPROGRAMS. COMBINE SEVERAL SUBPROGRAMS OR CHANGE LOGIC TO ELIMINATE A FEW.

BA049 **A SUB STATEMENT OCCURRED BEFORE AN END**
SUBPROGRAMS MUST OCCUR AFTER THE MAIN PROGRAM. THIS MEANS THAT THEY MUST IMMEDIATELY FOLLOW AN END STATEMENT OR ANOTHER SUBPROGRAMS SUBEND STATEMENT.

BA050 **STATEMENT FOLLOWING END/SUBEND NOT SUB/REM**
THE ONLY PERMISSIBLE STATEMENT FOLLOWING AN END STATEMENT IS A SUB OR REM STATEMENT. CORRECT AND RERUN.

BA051 **SECOND DEFINITION OF SUB — DEFINITION IGNORED**
TWO SUBPROGRAMS WITH THE SAME NAME HAVE BEEN ENCOUNTERED DURING THE COMPILATION PROCESS. THE SECOND SUBPROGRAM WILL BE EQUALED. THE SECOND SUBPROGRAM MAY HAVE BEEN FOUND IN A LIBRARY ELEMENT AS A RESULT OF A LIBRARY SEARCH. THIS IS A NON-FATAL ERROR.

Error Message/HELP Command Message

BA052 **SUB: FNX PRECEDES "CALL"**
A SUB STATEMENT DECLARING A PASSED FUNCTION CANNOT OCCUR BEFORE THE STATEMENT THAT CALLS IT (AND DEFINES THE FUNCTION PARAMETERS). RELOCATE THE SUBPROGRAM SO THAT IT OCCURS AFTER AT LEAST ONE STATEMENT THAT CALLS IT.

BA053 **FUNCTION EXPECTED IN CALL OR SUB LINE**
A PREVIOUS CALL STATEMENT PASSED A FUNCTION REFERENCE. THIS CALL DID NOT PASS A FUNCTION. THE PARAMETER TYPES MUST REMAIN THE SAME. RESOLVE THE CONFLICT AND RERUN THE PROGRAM.

BA054 **"SUBEND" OR SUBEXIT" NOT IN A SUB**
A SUBEND OR SUBEXIT WAS ENCOUNTERED THAT WAS NOT IN A SUBPROGRAM. THE SUBEND MUST BE THE LAST STATEMENT IN A SUBPROGRAM.

BA055 **"SUBEXIT" NOT ALLOWED IN FUNCTION DEFINITION**
A SUBEXIT STATEMENT WAS ENCOUNTERED WITHIN A MULTILINE FUNCTION DEFINITION. IT CAN ONLY BE ISSUED FROM THE SUBPROGRAM LEVEL.

BA056 **"FNEND" STATEMENT MISSING**
A USER DEFINED MULTILINE FUNCTION EXISTS IN THE PROGRAM WITHOUT A CLOSING FNEND STATEMENT. LOCATE THE FUNCTION AND INSERT THE STATEMENT.

BA057 **FUNCTION HAS NOT BEEN DEFINED**
THE FUNCTION REFERENCED ON THE LINE IN ERROR HAS NOT BEEN DEFINED. DEFINE THE FUNCTION OR REMOVE THE REFERENCE TO IT AND RERUN.

BA058 **LIMIT OF 4 "LIBRARY" STATEMENTS EXCEEDED**
BASIC WILL SEARCH AT MOST FOUR LIBRARIES FOR SUBPROGRAMS; THE PROGRAM HAS ATTEMPTED TO USE MORE THAN FOUR.

BA059 **TIME UP — PROGRAM LOOPING**
THE TIME LIMIT SPECIFIED IN THE TIME STATEMENT HAS BEEN EXCEEDED. IT MAY BE LOOPING OR IT MAY REQUIRE MORE TIME.

BA060 **RETURN WITHOUT MATCHING GOSUB CALL**
THE PROGRAM HAS ATTEMPTED TO RETURN FROM A SUBROUTINE THAT WAS NOT CALLED BY A GOSUB STATEMENT.

BA061 **EXPRESSION OUT OF COMPUTED GOTO RANGE**
THE CALCULATED EXPRESSION IS NOT A VALID NUMBER FOR THIS COMPUTED GOTO. IT IS EITHER TOO LARGE OR NON-POSITIVE. THE COUNT OF LINE NUMBERS IN THE STATEMENT DETERMINES THE LARGEST VALUE THE EXPRESSION MAY HAVE.

BA062 **EXECUTION STOPPED AT LINE XXXXX**
A STOP STATEMENT HAS BEEN ENCOUNTERED OR AN ERROR DETECTED AT THE LINE NUMBER GIVEN BY XXXXX.

BA063 **EXECUTION PAUSED AT LINE XXXXX CONTINUE (Y,N)**
A PAUSE STATEMENT HAS BEEN ENCOUNTERED AT LINE XXXXX. ANSWER "YES" TO CONTINUE EXECUTION; ANSWER "NO" TO TERMINATE THE PROGRAM.

BA064 **ACTIVE SUBROUTINES EXCEED 16 LEVELS**
A MAXIMUM OF 16 LEVELS OF SUBPROGRAM CALLS MAY BE ISSUED. INVESTIGATE FOR A POSSIBLE PROGRAM LOOP.

BA065 **#0 INVALID ON CHAIN**
CHANNEL 0, THE TERMINAL, MAY NOT BE USED AS THE FILE FROM WHICH THE CHAINED PROGRAM CAN BE READ. A DATA MANAGEMENT, TEMPORARY, OR LIBRARY FILE MUST BE USED.

BA066 **CHAIN ERROR — INVALID NAME OR PASSING BAD FILE**
THERE ARE TWO POSSIBLE CAUSES FOR THIS ERROR. THE LIBRARY ELEMENT SPECIFIED IN THE CHAIN STATEMENT DOES NOT EXIST, OR ONE OF THE CHANNEL NUMBERS OF FILES TO BE PASSED TO THE NEXT PROGRAM SEGMENT IS INVALID.

BA067 **ERROR ON READ FROM FILE (INVALID NUMBER)**
A READ STATEMENT ATTEMPTED TO READ A NUMERIC VARIABLE. THE RECORD THAT WAS READ DID NOT CONTAIN NUMERIC DATA.

BA068 **INPUT DATA INCORRECT, RE-ENTER**
THE DATA ENTERED FOR AN INPUT STATEMENT DOES NOT MATCH THE DATA TYPES REQUIRED FOR THE PROGRAM. THE ENTIRE LINE MUST BE RE-ENTERED. THIS MESSAGE COULD ALSO BE CAUSED BY TOO MUCH OR TOO LITTLE DATA IN THE INPUT RESPONSE.

BA069 **INVALID TAB EXPRESSION FOR PRINTING**
THE ARGUMENT OF THE TAB FUNCTION WAS LESS THAN ONE.

BA070 **PRINT TO FILE > MARGIN SIZE**
THE PROGRAM ATTEMPTED TO PRINT A STRING, NUMBER, OR USING STRING WITH A LENGTH GREATER THAN THE CURRENT MARGIN SETTING. CHANGE THE MARGIN SIZE OR REDUCE THE LENGTH OF THE EXPRESSION PRINTED.

BA071 **INVALID OPERATION FOR FILE TYPE**
THE OPERATION TO BE PERFORMED AGAINST THE FILE CONFLICTS WITH THE FILE TYPE.

BA072 **SET MARGIN FOR DMS FILE NOT AT RECORD 0**
A MARGIN STATEMENT WAS ISSUED AGAINST A DATA MANAGEMENT FILE WHILE IT STILL HAS DATA IN IT. THE MARGIN STATEMENT MAY ONLY BE USED WHEN THE FILE IS EMPTY.

BA073 **INVALID MARGIN SIZE**
THE MARGIN EXPRESSION SPECIFIED ON THE FLAGGED STATEMENT RESULTED IN A NUMBER LESS THAN 0 OR GREATER THAN 4095. THIS ERROR COULD ALSO HAVE RESULTED FROM ATTEMPTING TO SET THE SIZE OF THE MARGIN GREATER THAN THE LIMIT FOR THE FILE TYPE.

BA074 **OPERATION NOT PERMITTED TO FILE**
THE OPERATION TO BE PERFORMED AGAINST THE FILE CONFLICTS WITH THE FILE TYPE.

Error Message/HELP Command Message

BA075 **ATTEMPTED TO RESET FILE BEYOND EOF OR NEGATIVE**
THE RESET STATEMENT MAY NOT REPOSITION THE FILE BEYOND THE CURRENT END-OF-FILE POINTER. THE RECORD NUMBER SPECIFIED MUST BE POSITIVE.

BA076 **ATTEMPT TO TEST END OR MORE ON RANDOM FILE**
THE IF-END OR IF-MORE FORMATS MAY ONLY BE USED AGAINST TERMINAL FORMAT FILES. CHECK THE FILE TYPE REFERENCED BY THIS STATEMENT.

BA077 **INSUFFICIENT DATA TO READ**
ALL DATA STATEMENTS IN THE PROGRAM HAVE BEEN USED, YET THE PROGRAM ATTEMPTED TO REQUEST ADDITIONAL DATA.

BA078 **MORE THAN 32 FILES OPEN**
BASIC DOES NOT SUPPORT THE CONCURRENT USE OF MORE THAN 32 TEMPORARY AND/OR LIBRARY FILES PER USER. THIS PROGRAM HAS EXCEEDED THE LIMIT.

BA079 **CHANNEL NUMBER INVALID IN FILE STATEMENT**
THE CHANNEL-SETTER USED IN THE FILE STATEMENT RESULTS IN A CHANNEL NUMBER THAT IS NOT IN THE RANGE OF 1 TO 4095. CHANNEL 0 CANNOT BE DEFINED BY A FILE STATEMENT.

BA080 **NO FORMAT STRING DEFINED IN USING STRING**
THE USER PROGRAM ATTEMPTED TO PRINT A VARIABLE USING A FORMAT STRING THAT DOES NOT CONTAIN ANY FORMAT STRING CHARACTERS.

BA081 **NULL USING STRING NOT ALLOWED**
THE USING STRING SPECIFIED IN THE PRINT STATEMENT IS A NULL STRING. DEFINE THE VARIABLE AND RERUN.

BA082 **INVALID FIELD DESCRIPTOR, EXPECTING <,>**
THE USER PROGRAM ATTEMPTED TO PRINT A STRING VARIABLE WITH A NUMERIC FORMAT. CORRECT THE PROGRAM AND RERUN.

BA083 **INVALID FIELD DESCRIPTOR, EXPECTING $,+,—**
THE USER PROGRAM ATTEMPTED TO PRINT A NUMERIC VARIABLE WITH A STRING FORMAT. CORRECT THE PROGRAM AND RERUN.

BA084 **INVALID EXPONENT FIELD IN USING STRING**
AN EXPONENT FIELD MUST CONSIST OF EXACTLY FIVE UP ARROWS, AND CANNOT BE FOLLOWED BY A PLACE HOLDER NUMBER. CORRECT PROGRAM AND RERUN.

BA085 **DIVISION BY ZERO, EXECUTION CONTINUES**
THE PROGRAM HAS ATTEMPTED A DIVISION BY ZERO. THE ALGEBRAIC RESULT OF DIVISION BY ZERO IS UNDEFINED; HOWEVER, EXECUTION CONTINUES USING A HIGH VALUE.

BA086 **EXPONENT OVERFLOW, EXECUTION CONTINUES**
*THE RESULT (OR INTERMEDIATE RESULT) OF A COMPUTATION HAS EXCEEDED THE LARGEST NUMBER THE HARDWARE IS CAPABLE OF HANDLING. THIS NUMBER IS APPROXIMATELY 10**75. MACHINE INFINITY IS SUPPLIED AND EXECUTION CONTINUES.*

BA087 **BASIC FILE NOT OPEN OR NO DATA STATEMENTS**
THE CHANNEL NUMBER REFERENCED BY THE FLAGGED STATEMENT HAS NOT BEEN OPENED BY A FILE STATEMENT. CHECK THE CHANNEL SETTER FOR A VALID FILE, OR ISSUE A FILE STATEMENT FOR THE CHANNEL TO BE USED. THIS ERROR CAN ALSO RESULT WHEN READ STATEMENTS ARE ISSUED AND NO DATA ARE PRESENT.

BA088 **RENAME ERROR**
THE STRING EXPRESSION USED TO SUPPLY THE NEW FILE NAME DOES NOT CONTAIN A VALID FILE PARAMETER OR TEMPORARY FILE NAME. THIS ERROR MAY ALSO BE THE RESULT OF ATTEMPTING TO RENAME A DATA MANAGEMENT FILE.

BA089 **FILE IS NOT A LIBRARY FILE**
THE FILE SPECIFIED BY THE COMMAND IS NOT A LIBRARY FILE OR HAS NOT BEEN INITIALIZED BY THE LIBRARIAN. HAVE THE SYSTEM LIBRARIAN PREPARE THE FILE, AND BE SURE YOU ARE USING THE CORRECT FILE.

BA090 ***FATAL WORK SPACE FILE DOPEN ERROR**
BASIC CANNOT ALLOCATE THE DISK WORKSPACE. BASIC TASK WILL BE ABNORMALLY TERMINATED.

BA091 ***FATAL WORK SPACE FILE DCLOSE ERROR**
AN I/O ERROR HAS OCCURRED WHILE CLOSING THE DISK WORKSPACE. BASIC TASK WILL BE ABNORMALLY TERMINATED.

BA092 **WORK SPACE FILE DMSEL/DMINP ERROR**
AN I/O ERROR HAS OCCURRED WHILE READING FROM THE DISK WORKSPACE.

BA093 **WORK SPACE FILE DMINP/DMUPD/DMOUT ERROR**
AN I/O ERROR HAS OCCURRED WHILE WRITING TO THE DISK WORKSPACE.

BA094 **WORK SPACE FILE DMINP/DMDEL ERROR**
AN I/O ERROR HAS OCCURRED WHILE DELETING FROM THE DISK WORKSPACE.

BA095 ***FATAL TERMINAL FILE DMOUT ERROR**
AN I/O ERROR HAS OCCURRED WHILE WRITING TO THE TERMINAL FILE. BASIC TASK WILL BE ABNORMALLY TERMINATED.

BA096 ***FATAL FILE DMINP ERROR**
AN I/O ERROR HAS OCCURRED WHILE READING FROM THE TERMINAL FILE. BASIC TASK WILL BE ABNORMALLY TERMINATED.

BA097 **LIBRARY FILE DCLOSE ERROR**
AN I/O ERROR HAS OCCURRED WHILE CLOSING THE LIBRARY FILE.

BA098 **LIBRARY FILE READ ERROR**
AN I/O ERROR HAS OCCURRED WHILE READING FROM THE LIBRARY FILE.

BA099 **LIBRARY FILE WRITE ERROR**
AN I/O ERROR HAS OCCURRED WHILE WRITING TO THE LIBRARY FILE.

Error Message/HELP Command Message

BA100 **I/O ERROR WHILE ACCESSING V.T.O.C.**
AN I/O ERROR HAS OCCURRED WHILE ACCESSING THE VTOC FOR THE DISK VOLUME SPECIFIED. RETRY OR INVESTIGATE FOR POSSIBLE HARDWARE PROBLEM.

BA101 **I/O ERROR ON WRITE TO FILE**
AN I/O ERROR HAS OCCURRED WHILE WRITING TO A DATA MANAGEMENT FILE. INVESTIGATE FOR POSSIBLE HARDWARE PROBLEM OR RETRY THE PROGRAM.

BA102 **DATA FILE FULL, DATA NOT ADDED**
CANNOT ACQUIRE ANY ADDITIONAL SPACE.

BA103 **INVALID BLOCK SIZE OR RECORD SIZE**
BASIC CANNOT PROCESS THE FILE DUE TO A CONFLICT WITH THE BLOCK OR RECORD SIZE FOR THIS FILE. IF THE FILE ALREADY EXISTS, CHECK THAT THE BLOCK SIZE OR RECORD SIZE IS NOT 0 OR GREATER THAN 65K.

BA104 **ERROR PROCESSING USER FILE LABEL**
THE FILE BEING ACCESSED CONTAINS USER FILE LABELS. THESE CANNOT BE PROCESSED BY BASIC.

BA105 **INVALID KEY LENGTH**
FILES CONTAINING KEYS CANNOT BE PROCESSED BY BASIC.

BA106 **INTERNAL ERROR IN FILE ACCESS ROUTINE**
AN INTERNAL ERROR HAS BEEN DETECTED IN THE FILE ACCESS ROUTINE IN BASIC.

BA107 **I/O ERROR WHILE READING DATA FILE**
AN I/O ERROR HAS OCCURRED WHILE READING FROM THE DATA FILE.

BA108 **I/O ERROR WHILE WRITING DATA FILE**
AN I/O ERROR HAS OCCURRED WHILE WRITING TO THE DATA FILE.

BA109 ***FATAL GETBUF/FREEBUF ERROR**
BASIC COULD NOT ACQUIRE SYSTEM BUFFER POOL. BASIC TASK WILL ABNORMALLY TERMINATE.

BA110 **SYSTEM COMMAND REJECTED**
BASIC SYSTEM COMMAND HAS BEEN REJECTED. CHECK THE SYSTEM COMMAND STRING.

BA111 **LIBRARY FILE DOPEN ERROR**
AN I/O ERROR HAS OCCURRED WHILE OPENING THE LIBRARY FILE.

BA112 **BATCH END-OF-DATA REACHED**
BASIC PROGRAM RUNNING IN ENTER STREAM HAS ENCOUNTERED AN END-OF-DATA CONDITION. RERUN PROGRAM WITH ENOUGH DATA TO SATISFY INPUT REQUESTS.

BA113 **BASIC TASK NORMAL TERMINATION**
BASIC TASK HAS TERMINATED NORMALLY. THIS MESSAGE IS INFORMATIONAL ONLY.

BA114 **ELEMENT IS NOT IN THE LIBRARY FILE**
THE ELEMENT REQUESTED BY THE COMMAND IS IN THE FILE SPECIFIED. CHECK THE SPELLING OF THE PROGRAM NAME AND VERIFY THAT THE PROGRAM IS ON THE FILE. ALSO BE SURE THE CORRECT MODULE TYPE HAS BEEN USED (P FOR PROCS).

BA115 **TANGENT/COTANGENT OUT OF RANGE**
THE RESULT OF A TAN OR COT FUNCTION EVALUATION CAUSED AN OVERFLOW CONDITION. MACHINE INFINITY IS SUPPLIED AND EXECUTION CONTINUES.

BA116 **ARGUMENT TOO LARGE FOR EXP(X) FUNCTION**
A VALUE HAS BEEN USED WITH THE EXPONENTIAL FUNCTION WHICH WILL PRODUCE A RESULT GREATER THAN THE HARDWARE IS CAPABLE OF HANDLING. THE MAXIMUM POSSIBLE VALUE FOR THE EXP ARGUMENT IS APPROXIMATELY 174.6.

BA117 **NO MEMORY AVAILABLE FOR FILE I/O BUFFER**
AN AREA OF MAIN STORAGE COULD NOT BE ACQUIRED FOR THE DATA FILE I/O BUFFER.

BA118 **SOURCE MODULE NOT SAVED — TERMINATE (Y,N)?**
SOURCE PROGRAM REMAINS IN THE DISK WORKSPACE WHEN A BYE COMMAND IS ENTERED. IF THE SOURCE MODULE NEEDS TO BE SAVED, ENTER "N" FOLLOWED BY THE APPROPRIATE SAVE COMMAND. IF IT IS NOT NEEDED, ANSWER "Y" TO TERMINATE THE BASIC SESSION.

HELP Command Message for Syntax Error

BA119 **RECURSIVE FUNCTION CALLS NOT ALLOWED**
A FUNCTION OR SUBPROGRAM MAY NOT CALL ITSELF DIRECTLY OR INDIRECTLY (VIA ANOTHER FUNCTION OR SUBPROGRAM).

BA126 **MG#BSBVD**
EACH BASIC STATEMENT MUST BEGIN WITH A VALID LINE NUMBER. THIS LINE NUMBER MUST BE IN THE RANGE 1 TO 99999, AND MUST NOT CONTAIN A DECIMAL POINT OR AN EXPONENT. ENTER A VALID LINE NUMBER.

BA127 **MG#INSVD**
BASIC INSTRUCTION EXPECTED. VALID INSTRUCTIONS WHICH MAY BEGIN A STATEMENT ARE:

LET	*IF*	*GO*	*ON*	*RETURN*
PAUSE	*STOP*	*END*	*FOR*	*NEXT*
DEF	*FNEND*	*SUB*	*SUBEND*	*CALL*
LIBRARY	*REM*	*DATA*	*FILE*	*RESET*
RESTORE	*READ*	*INPUT*	*LINPUT*	*WRITE*
PRINT	*MARGIN*	*SCRATCH*	*CHAIN*	*CHANGE*
RANDOMIZE	*DIM*	*MAT*	*TIME*	*SYSTEM*
SUBEXIT				

BA128 **MG#COPEX**
A COMPARISON OPERATOR IS EXPECTED AT THIS POINT. VALID COMPARISON OPERATORS ARE : =,<,>,=>,>=, =<,<=,<>

BA129 **MG#MIXMD**
MIXED MODE COMPARISON IS INVALID. A NUMERIC EXPRESSION MAY ONLY BE COMPARED TO ANOTHER NUMERIC EXPRESSION, OR A STRING EXPRESSION TO ANOTHER STRING.

BA130 **MG#ISSTE**
AN IF STATEMENT MUST END WITH A "GOTO LINE-NUM", "GOSUB LINE-NUM", OR "THEN LINE-NUM".

BA131 **MG#GOINS**
THE "GO" INSTRUCTION MUST BE FOLLOWED BY A "TO" OR A "SUB", AS IN "GOTO" OR "GOSUB".

BA132 **MG#LNENB**
A LINE NUMBER IS EXPECTED HERE.

BA133 **MG#GOEPN**
THE WORD "GOTO" OR "GOSUB" IS EXPECTED NEXT. THE GENERAL FORMAT OF AN "ON" STATEMENT IS:
ON NUM-EXPR GOTO LINE-NUM,LINE-NUM,...
ON NUM-EXPR GOSUB LINE-NUM,LINE-NUM,...

BA134 **MG#EDOPR**
AN END OR MORE OPTION REQUIRES THAT A CHANNEL SETTER FOLLOW. ENTER "#N" TO COMPLETE THE "END" OR "MORE" OPTION.

BA135 **MG#LNBSR**
A LINE NUMBER OR SERIES OF LINE NUMBERS IS EXPECTED HERE. IF A SERIES OF LINE NUMBERS ARE PRESENT, THEY MUST BE SEPARATED BY COMMAS. THE GENERAL FORMAT OF AN "ON" STATEMENT IS:
ON NUM-EXPR GOTO LINE-NUM,LINE-NUM,...
ON NUM-EXPR GOSUB LINE-NUM,LINE-NUM,...

BA136 **MG#MIXAS**
MIXED MODE ASSIGNMENTS ARE NOT PERMITTED. AN EXPRESSION WITH A NUMERIC VALUE MAY ONLY BE ASSIGNED TO A NUMERIC VARIABLE, AND EXPRESSIONS WITH STRING VALUES MAY ONLY BE ASSIGNED TO STRING VARIABLES. STRING VARIABLES ARE DISTINGUISHED FROM NUMERIC VARIABLES BY THE PRESENCE OF A DOLLAR SIGN: A$,Z3$ — STRING AND A,Z3 — NUMERIC.

BA137 **MG#EQUSN**
AN EQUAL SIGN IS REQUIRED BETWEEN VARIABLES, OR BETWEEN VARIABLES AND THE ASSIGNED EXPRESSION:
LET VAR1=VAR2=...=VARN=EXPRESSION

BA138 **MG#EXPAS**
THE EXPRESSION TO BE ASSIGNED TO THIS VARIABLE IS NOT PRESENT OR IS INCOMPLETE. AN EXPRESSION OR ANOTHER VARIABLE MUST FOLLOW THE EQUAL SIGN.

BA139 **MG#EQSEP**
AN EQUAL SIGN IS EXPECTED HERE. A POSSIBLE CAUSE FOR THIS ERROR COULD BE AN ATTEMPT TO ASSIGN A VALUE TO AN EXPRESSION. VALUES MAY ONLY BE ASSIGNED TO VARIABLES OR FUNCTION NAMES.

BA140 **MG#SPNMV**
A SIMPLE NUMERIC VARIABLE NAME, FOLLOWED BY AN ASSIGNMENT IS EXPECTED AFTER THE "FOR" INSTRUCTION. THE GENERAL FORMAT OF THE "FOR" INSTRUCTION IS:
FOR VAR=EXPR TO EXPR
FOR VAR=EXPR TO EXPR STEP EXPR

BA141 **MG#ONLAL**
THE ONLY VALID INSTRUCTIONS AT THIS POINT ARE "TO" OR "STEP". THIS ERROR COULD ALSO BE THE RESULT OF NOT SPECIFYING A "TO" EXPRESSION. THE GENERAL FORMAT OF THE "FOR" INSTRUCTION IS:
FOR VAR=EXPR TO EXPR
FOR VAR=EXPR TO EXPR STEP EXPR

BA142 **MG#SDTOA**
THIS IS THE SECOND TIME "TO" APPEARED IN THIS STATEMENT. IT IS ALLOWED ONLY ONCE. THE GENERAL FORMAT OF THE 'FOR" STATEMENT IS:
FOR VAR=EXPR TO EXPR
FOR VAR=EXPR TO EXPR STEP EXPR

HELP Command Message for Syntax Error

BA143 **MG#SDSTA**
THIS IS THE SECOND TIME "STEP" APPEARED IN THIS STATEMENT. IT IS ALLOWED ONLY ONCE. THE GENERAL FORMAT OF THE "FOR" STATEMENT IS:
FOR VAR=EXPR TO EXPR
FOR VAR=EXPR TO EXPR STEP EXPR

BA144 **MG#SMPRQ**
A SIMPLE NUMERIC VARIABLE NAME IS EXPECTED HERE. THE GENERAL FORMAT OF THE NEXT STATEMENT IS:
NEXT VAR

BA145 **MG#FUNCD**
A FUNCTION DEFINITION MUST BEGIN WITH THE FUNCTION NAME. A FUNCTION NAME BEGINS WITH "FN" IMMEDIATELY FOLLOWED BY A LETTER.
THE GENERAL FORMAT FOR A DEFINITION IS:
DEF FCN-NAME
DEF FCN-NAME(PARAM-LIST)
DEF FCN-NAME LOCAL-LIST
DEF FCN-NAME (PARAM-LIST) LOCAL-LIST
AN ASSIGNMENT MAY BE MADE AT THE END OF ANY OF THESE FORMATS, FOR EXAMPLE:
DEF FCN-NAME(PARAM-LIST)=EXPR
** PARAM-LIST SPECIFIES PARAMETERS TO BE PASSED TO THE FUNCTION.*
** LOCAL-LIST SPECIFIES LOCAL VARIABLE NAMES.*
** IF THE FUNCTION CAN BE DEFINED IN ONE LINE, AN "FNEND" STATEMENT IS NOT NEEDED.*

BA146 **MG#STMEV**
A STATEMENT MUST END HERE, OR A VALUE ASSIGNMENT MUST BE MADE. REFER TO "HELP BA145" FOR A GENERAL DESCRIPTION OF A FUNCTION DEFINITION.

BA147 **MG#PRLPR**
THE PRESENCE OF A LEFT PARENTHESIS HERE MEANS YOU ARE TRYING TO LIST PASSED PARAMETERS FOR THIS FUNCTION. THESE ARE SPECIFIED BY A LIST OF VARIABLE NAMES (A,B2,C$,Z1$,F,...) SEPARATED BY COMMAS. THE LIST MUST TERMINATE WITH A RIGHT PARENTHESIS. REFER TO "HELP BA145" FOR A GENERAL DESCRIPTION OF A FUNCTION DEFINITION.

BA148 **MG#CMAFN**
A COMMA AFTER THE FUNCTION NAME OR THE LIST OF PARAMETERS INDICATES THAT YOU ARE TRYING TO STATE THE NAMES OF THE LOCAL VARIABLES, SPECIFIED BY A SERIES OF VARIABLE NAMES SEPARATED BY COMMAS. REFER TO "HELP BA145" FOR A GENERAL DESCRIPTION OF A FUNCTION DEFINITION.

BA149 **MG#SBDST**
A SUBPROGRAM DEFINITION MUST BEGIN WITH A STRING CONSTANT STATING THE NAME OF THE SUBPROGRAM. THIS IS FOLLOWED BY A COLON, AND THEN AN OPTIONAL LIST OF PASSED PARAMETERS:
SUB-STRING:FUNCTION,FUNCTION
SUB-STRING:FILE-NUM,FILE-NUM,...
SUB-STRING:MATRIX-NAME,MATRIX-NAME
SUB-STRING:VAR,VAR

BA150 **MG#FNFMM**
ANOTHER FILE-NUMBER, FUNCTION-NAME, MATRIX, OR VARIABLE-NAME IS EXPECTED AFTER THE COMMA. IF NO MORE ARE TO BE SPECIFIED, REMOVE THE EXTRA COMMA; OTHERWISE DEFINE THE EXTRA NAME.

BA151 **MG#STEDC**
THE STATEMENT MUST END HERE, OR A COMMA MUST BE USED TO SEPARATE THE LIST OF NAMES.

BA152 **MG#MRDEP**
A MATRIX DEFINITION IS EXPECTED HERE. THIS MAY BE EITHER A VECTOR, SPECIFIED BY A VECTOR NAME FOLLOWED BY A LEFT-PAREN AND RIGHT-PAREN: V()— OR AN ARRAY, SPECIFIED BY A MATRIX NAME FOLLOWED BY A LEFT-PAREN, A COMMA, AND A RIGHT-PAREN: A(,). TO CORRECT THIS, ENTER A COMMA OR A ")".

BA153 **MG#CALST**
THE "CALL" STATEMENT MUST BEGIN WITH A STRING CONSTANT STATING THE NAME OF THE SUBPROGRAM TO BE CALLED. THIS IS FOLLOWED BY A COLON, AND THEN A LIST OF VARIABLES, EXPRESSIONS, FUNCTIONS, OR MATRICIES OR FILES TO BE PASSED. THE GENERAL FORMAT IS:

CALL-STRING:	*#N*	*,#N*	*,...ETC*
	VAR	*VAR*	
	FUNCT	*FUNCT*	
	EXPR	*EXPR*	
	MATRIX	*MATRIX*	

BA154 **MG#STMEC**
THE CALL STATEMENT MUST BEGIN WITH A STRING CONSTANT STATING THE NAME OF THE SUB-PROGRAM TO BE CALLED. THIS IS FOLLOWED BY A COLON, AND THEN A LIST OF PASSED PARAMETERS. THE GENERAL FORMAT OF THE CALL STATEMENT IS:

CALL-STRING—	*#N*	*,#N*	
	VAR	*VAR*	
	FUNCTION	*FUNCTION*	
	EXPR	*EXPR*	.....
	MATRIX	*MATRIX*	

BA155 **MG#FNIDT**
A FUNCTION NAME HAS JUST BEEN DETECTED. IT COULD BE AN EXPRESSION CONTAINING A FUNCTION VALUE TO BE PASSED, IN WHICH CASE IT SHOULD BE FOLLOWED BY A LEFT PAREN. IT COULD ALSO BE THE NAME OF A FUNCTION TO BE PASSED, IN WHICH CASE IT MUST BE THE LAST ITEM IN THE STATEMENT, OR FOLLOWED BY A COMMA. THE ERROR WAS CAUSED BY THE FUNCTION NAME NOT BEING AT THE END OF A LINE, OR NOT FOLLOWED BY A LEFT PAREN OR COMMA.

BA156 **MG#MXIDT**
A MATRIX REFERENCE HAS JUST BEEN DETECTED IN WHICH AN ENTIRE MATRIX IS TO BE PASSED TO A SUBPROGRAM. TO DO SO, THE COMMA MUST BE FOLLOWED BY A RIGHT-PAREN - A3(,) OR F$(,).

HELP Command Message for Syntax Error

BA157 **MG#LIBST**
A "LIB" STATEMENT IS COMPOSED OF A LIST OF LIBRARIES TO BE SEARCHED FOR SUBPROGRAMS. THESE ARE STATED AS STRING CONSTANTS, SEPARATED BY COMMAS. THE FORMAT FOR THIS STATEMENT IS:
LIB STRING,STRING,...

BA158 **MG#CHINP**
THE CHANNEL SETTER (#) IMPLIES THAT YOU ARE TRYING TO DEFINE A FILE TO BE PASSED TO THIS SUBPROGRAM. IT MUST BE FOLLOWED BY A NUMERIC CONSTANT TO BE CORRECT — #3, #17.5, ETC.

BA159 **MG#CHEPC**
A CHANNEL EXPRESSION IS COMPOSED AS FOLLOWS:
#NUMERIC-EXPR:
WHERE NUMERIC-EXPR IS THE CHANNEL TO BE REFERENCED.

BA160 **MG#FLSTD**
A "FILE" STATEMENT DEFINES THE FILE NAME AND ASSOCIATED CHANNEL NUMBER TO BE USED WITH IT:
FILE #N:"EXTERNAL-FILE-NAME"
WHERE #N IS THE ASSOCIATED CHANNEL NUMBER. SEE UP-9168 FOR DETAILS ON "EXTERNAL-FILE-NAME".

BA161 **MG#RSSTC**
A "RESET" STATEMENT MAY OPTIONALLY CONTAIN A CHANNEL SETTER, AND IF ONE IS CODED, A NUMERIC FILE POSITION MAY ALSO BE SPECIFIED. THE STATEMENT MUST EITHER END HERE, OR A CHANNEL EXPRESSION MUST FOLLOW.
THE GENERAL FORMAT FOR A "RESET" STATEMENT IS:
RESET
RESET #N
RESET #N:NUMERIC-EXPR

BA162 **MG#CHSRS**
THE CHANNEL STATEMENT IN A "RESET" STATEMENT MUST EITHER END THE STATEMENT OR BE FOLLOWED BY A COLON (:) AND ANOTHER EXPRESSION. THE GENERAL FORMAT OF THE RESET STATEMENT IS:
RESET
RESET #N
RESET #N:NUMERIC-EXPR

BA163 **MG#RIWST**
A "READ" OR "WRITE" STATEMENT IS COMPOSED OF AN OPTIONAL CHANNEL SETTER FOLLOWED BY A LIST OF VARIABLE NAMES (IN THE CASE OF A "READ"), OR A LIST OF EXPRESSIONS (IN THE CASE OF A "WRITE"). EACH VARIABLE OR EXPRESSION MUST BE SEPARATED BY A COMMA ("READ"), OR COMMA OR SEMICOLON ("WRITE"). THE GENERAL FORMAT IS:
INPUT VAR1,VAR2,...,VARN
INPUT #N:VAR1,VAR2,...,VARN
READ VAR1,VAR2,...,VARN
READ #N: VAR1,VAR2,...,VARN
WRITE VAR1,VAR2,...,VARN
WRITE VAR1;VAR2;...;VARN
ETC.

BA164 **MG#ILSYM**
AN ILLEGAL SYMBOL HAS BEEN FOUND IN THE INPUT. THIS COULD BE THE RESULT OF FINDING A CHARACTER WHICH IS NOT IN THE BASIC CHARACTER SET, OR AN INVALID VARIABLE NAME OR NUMERIC CONSTANT. A NUMERIC CONSTANT IS MADE UP OF A FRACTIONAL PART AND AN OPTIONAL EXPONENT PART. THE FRACTIONAL PART MAY CONTAIN AT MOST ONE DECIMAL POINT, AND MUST BEGIN WITH A DIGIT OR A DECIMAL POINT. THE EXPONENT PART MUST CONSIST OF THE LETTER E FOLLOWED OPTIONALLY BY A SIGN AND ONE OR TWO DIGITS. THE EXPONENT MUST NOT BE GREATER THAN E70 OR LESS THAN E—70.

BA165 **MG#TABPU**
"TAB" MAY NOT BE USED WITH "PRINT USING". ANY TAB EXPRESSION IS INVALID WHILE IN PRINT USING MODE.

BA166 **MG#PULST**
A PRINT USING LIST HAS BEEN FOUND IMMEDIATELY AFTER ANOTHER PRINT USING LIST, WITHOUT A TERMINATOR FOR THE FIRST LIST. A "PRINT USING" LIST CONSISTS OF THE KEYWORD "USING", FOLLOWED BY A STRING EXPRESSION, A LIST OF VARIABLES TO BE PRINTED, AND FINALLY A SEMICOLON OR END-OF-LINE. ANOTHER "PRINT USING" LIST MAY NOT APPEAR UNTIL THE FIRST LIST IS TERMINATED. SEMICOLON IS EXPECTED HERE, THEN THE NEXT USING.

BA167 **MG#MARGN**
THE "MARGIN" STATEMENT IS EXPECTED TO END HERE. EXTRA CHARACTERS ARE PRESENT IN THE INPUT LINE. THE FORMAT FOR A MARGIN STATEMENT IS:
MARGIN NUMERIC-EXPR
MARGIN #N:NUMERIC-EXPR

BA168 **MG#SCRCH**
THE "SCRATCH" STATEMENT IS EXPECTED TO END HERE. EXTRA CHARACTERS ARE PRESENT IN THE INPUT LINE. THE FORMAT FOR A "SCRATCH" STATEMENT IS:
SCRATCH
SCRATCH #N

BA169 **MG#CHAIN**
THE "CHAIN" STATEMENT MUST END HERE, OR THE NEXT WORD MUST BE "WITH". THE GENERAL FORMAT OF A CHAIN STATEMENT IS:
CHAIN #N
CHAIN #N WITH #N,#N,...
CHAIN "EXTERNAL-FILE-NAME"
CHAIN "EXTERNAL-FILE-NAME" WITH "EXT-FIL-NAME",...

BA170 **MG#LSOFL**
WHEN A LIST OF FILES TO BE PASSED TO THE CHAINED PROGRAM IS SPECIFIED, EACH FILE NUMBER IN THE LIST MUST BE SEPARATED BY A COMMA.

HELP Command Message for Syntax Error

BA171 **MG#CHAIS**
IN THIS FORM OF THE "CHANGE" INSTRUCTION, A STRING IS BEING CONVERTED TO A NUMERIC ARRAY. FOLLOWING THE STRING EXPRESSION, THE KEYWORD "TO", AND THEN A NUMERIC MATRIX NAME CONSISTING OF A SINGLE LETTER IS REQUIRED. EITHER ENTER THE WORD "TO" OR A PROPER MATRIX NAME TO CORRECT THE ERROR.
THE GENERAL FORM OF THE "CHANGE" STATEMENT IS:
CHANGE STRING-EXPR TO LETTER
CHANGE STRING-EXPR TO LETTER BIT EXPR
CHANGE LETTER TO STRING-VAR
CHANGE LETTER TO STRING-VAR BIT EXPR

BA172 **MG#CGEIS**
IN THIS FORM OF THE "CHANGE" INSTRUCTION, A NUMERIC ARRAY IS BEING CONVERTED TO STRING. FOLLOWING THE NUMERIC ARRAY, THE KEYWORD "TO", AND THEN A STRING VARIABLE NAME ARE REQUIRED. EITHER ENTER THE KEYWORD "TO" OR A STRING VARIABLE NAME TO CORRECT THE ERROR.

BA173 **MG#EARDM**
EACH ARRAY TO BE DEFINED IN A "DIM" STATEMENT CONSISTS OF A SINGLE LETTER, OPTIONALLY FOLLOWED BY A DOLLAR SIGN ($) FOR STRING ARRAYS. THE VECTOR DEFINITION CONSISTS OF A LEFT-PAREN, AN INTEGER IN THE RANGE 1—99999, AND A RIGHT-PAREN. AN ARRAY DEFINITION CONSISTS OF A LEFT-PAREN, AN INTEGER, A COMMA, ANOTHER INTEGER, AND A RIGHT-PAREN. THE GENERAL FORMAT FOR A DIMENSION STATEMENT IS:
DIM VECTOR-NAME(INTEGER)
DIM ARRAY-NAME(INTEGER,INTEGER)
DIM STRING-NAME(INTEGER)...ETC
MORE THAN ONE ARRAY MAY BE DECLARED IN A DIM STATEMENT, BUT THE SAME NAME CANNOT BE DIMENSIONED MORE THAN ONCE IN ANY PROGRAM.

BA174 **MG#EITDM**
EITHER THE DIM STATEMENT MUST END HERE, OR A COMMA AND ANOTHER ARRAY DEFINITION MUST FOLLOW. MORE THAN ONE VECTOR OR ARRAY MAY BE DEFINED IN ONE DIM STATEMENT, BUT A NAME MAY NOT APPEAR IN MORE THAN ONE DIM PER PROGRAM.

BA175 **MG#MATMX**
A MATRIX STATEMENT MUST BEGIN WITH A MATRIX NAME, OR THE MATRIX INSTRUCTIONS READ, WRITE, INPUT, LINPUT, OR PRINT.

BA176 **MG#TIMST**
TIME STATEMENT REQUIRES POSITIVE INTEGER VALUE FOR A TIME LIMIT.

BA177 **MG#OPMIO**
OPERANDS FOR MATRIX I/O STATEMENTS MUST CONSIST OF A SINGLE LETTER OPTIONALLY FOLLOWED BY A DOLLAR SIGN ($) IN THE CASE OF A STRING MATRIX. FOR A MAT LINPUT STATEMENT, A STRING MATRIX NAME IS REQUIRED. A MATRIX NAME IS REQUIRED HERE.

BA178 **MG#MIOST**
MATRIX I/O STATEMENTS MAY INPUT OR OUTPUT MORE THAN ONE MATRIX IN A SINGLE STATEMENT. EACH MATRIX IN THE LIST TO BE PROCESSED MUST BE SEPARATED BY COMMAS AND THERE MUST BE NOTHING AFTER THE LAST MATRIX IN THE LIST. EITHER END THE STATEMENT HERE OR ENTER A COMMA AND ANOTHER NAME.

BA179 **MG#CMAUS**
A COMMA MUST APPEAR AFTER THE "USING" KEYWORD. ENTER A COMMA AND COMPLETE THE STATEMENT.

BA180 **MG#MTNMR**
A MATRIX NAME IS REQUIRED FOR THE "MAT PRINT" STATEMENT. THIS MAY BE A NUMERIC MATRIX (SINGLE LETTER), OR A STRING MATRIX (LETTER FOLLOWED BY A DOLLAR SIGN). ENTER THE NAME AND COMPLETE THE STATEMENT.

BA181 **MG#THEST**
THE STATEMENT MUST EITHER END HERE OR A COMMA OR SEMICOLON MUST BE TYPED TO CONTINUE THE LIST OF MATRIX NAMES. IF MORE THAN ONE MATRIX NAME IS DESIRED EACH NAME IN THE LIST MUST BE SEPARATED BY A COMMA OR SEMICOLON.

BA182 **MG#VARNM**
A VARIABLE NAME HAS BEEN FOUND WHICH INDICATES THAT A SCALAR RESULT IS REQUIRED. THIS FORMAT OF THE MAT STATEMENT REQUIRES TWO NUMERIC VECTORS BE MULTIPLIED TO GIVE A SCALAR RESULT.
THE FORMAT IS:
*MAT LETTER-NUMBER = VECT-NAME * VECT-NAME*
*EXAMPLE: MAT A1=B*C*

BA183 **MG#LTVSB**
THE LAST VALID SYMBOL FOUND IN THE INPUT LINE WAS A MATRIX NAME. THIS MUST BE FOLLOWED BY AN EQUAL SIGN, AND THEN EITHER A MATRIX COMPUTATION OR A FUNCTION ASSIGNMENT. IF THIS IS TO BE A MATRIX COMPUTATION, THEN A NUMERIC MATRIX NAME MUST FOLLOW THE EQUAL SIGN; OTHERWISE ONE OF THESE FUNCTIONS MUST BE USED: ZER, CON, IDN, INV, TRN.

BA184 **MG#MIVTM**
IN A "MAT TRN" OR "MAT INV", THE FUNCTION NAME MUST BE FOLLOWED BY A MATRIX NAME WITHIN PARENTHESES.

BA185 **MG#OSTFC**
THE ONLY MATRIX STRING FUNCTION AVAILABLE IS NUL$, AND ITS FORMAT IS:
MAT LETTER$=NUL$

BA186 **MG#OVDSB**
*THE ONLY VALID SYMBOLS WHICH MAY FOLLOW THE MATRIX NAME HERE ARE +, —, OR *:*
MAT LETTER = LETTER + LETTER
MAT LETTER = LETTER — LETTER
*MAT LETTER = LETTER * LETTER*

HELP Command Message for Syntax Error

BA187 **MG#NMTNM**
A NUMERIC MATRIX NAME MUST BE ENTERED HERE. THIS CONSISTS OF A SINGLE LETTER, AND MUST BE THE LAST SYMBOL IN THE STATEMENT.

BA188 **MG#MLINP**
THE VARIABLE LIST FOR A "MAT LINPUT" MUST CONTAIN ONLY STRING MATRIX NAMES CONSISTING OF A LETTER FOLLOWED BY A DOLLAR SIGN ($). EACH NAME IN THE LIST MUST BE SEPARATED BY A COMMA.

BA189 **MG#MTREP**
A MATRIX TRIMMER EXPRESSION BEGINS WITH A LEFT-PAREN, AND A NUMERIC EXPRESSION. IF THIS TRIMMER IS USED FOR A VECTOR, THEN A RIGHT-PAREN MUST TERMINATE THIS EXPRESSION. IF THIS IS TO TRIM AN ARRAY, THEN A COMMA AND ANOTHER NUMERIC EXPRESSION MUST PRECEDE THE RIGHT-PAREN. TO CORRECT THE ERROR, ENTER EITHER A COMMA OR A RIGHT-PAREN AND CONTINUE.

BA190 **MG#ADNCH**
THERE ARE ADDITIONAL CHARACTERS ON THIS LINE WHICH ARE NOT EXPECTED. THE STATEMENT MUST END HERE. IF THE ADDITIONAL CHARACTERS ARE TO BE A COMMENT OR REMARK, THEY MUST BE PREFIXED BY AN APOSTROPHE (').

BA191 **MG#SEQSB**
THE SEQUENCE OF SYMBOLS IN THE EXPRESSION IS NOT CORRECT. THE SYMBOL IN ERROR CANNOT FOLLOW THE SYMBOL SHOWN ABOVE. FOR EXAMPLE, TWO VARIABLE NAMES MAY NOT APPEAR TOGETHER, TWO OPERATORS MAY NOT APPEAR TOGETHER (A+/B), A STATEMENT MAY ONLY END IN A VARIABLE NAME, OR RIGHT PARENTHESIS, ETC.

BA192 **MG#EXPRE**
AN EXPRESSION IS EXPECTED HERE. STRING EXPRESSIONS CAN BEGIN WITH STRING LITERALS (QUOTE CLOSED-STRING-CHARS QUOTE — "STRING"), OR A STRING VARIABLE NAME (A$,W3$), OR A STRING FUNCTION NAME (FNT$,FNA$,ETC). NUMERIC EXPRESSIONS CAN BEGIN WITH A NUMBER, A NUMERIC VARIABLE, A NUMERIC FUNCTION, A LEFT-PAREN, A UNARY MINUS SYMBOL, OR A UNARY PLUS SYMBOL. NUMERIC VARIABLE AND FUNCTION ARE CONSTRUCTED THE SAME AS STRING VARIABLES AND FUNCTIONS, BUT WITHOUT THE DOLLAR SIGN.

BA193 **MG#UARMI**
*A UNARY MINUS SYMBOL IS NOT ALLOWED AT THIS POINT. AT THIS POINT IN THE EXPRESSION, OPERATORS (+, —, *, /, **) LEFT-PARENS, OR FUNCTION REFERENCES MAY NOT BE CODED HERE. A VARIABLE OR CONSTANT MAY BE CODED HERE.*

BA194 **MG#VRFCN**
A VARIABLE OR FUNCTION NAME IS NOT PERMITTED HERE. A CONSTANT (STRING OR NUMERIC) MAY BE USED.

BA195 **MG#NUMIT**
A NUMERIC ITEM IS EXPECTED HERE. STRING SYMBOLS ARE NOT PERMITTED. PERMISSIBLE NUMERIC EXPRESSION SYMBOLS ARE: NUMERIC VARIABLE NAMES, NUMERIC FUNCTION NAMES, NUMERIC CONSTANTS, OR NUMERIC OPERATORS.

BA196 **MG#STRIT**
A STRING ITEM IS EXPECTED HERE. NUMERIC SYMBOLS ARE NOT PERMITTED. POSSIBLE STRING EXPRESSION SYMBOLS ARE: STRING VARIABLE NAMES, STRING CONSTANTS, STRING FUNCTION NAMES, OR THE STRING OPERATOR FOR CONCATENATION (&).

BA197 **MG#STRCN**
A STRING CONSTANT IS NOT ALLOWED HERE. SYMBOLS WHICH MAY BE ALLOWED HERE ARE NUMERIC SYMBOLS, OR STRING VARIABLE OR FUNCTION NAMES.

BA198 **MG#EDIPL**
THE END OF THE INPUT LINE WAS DETECTED BEFORE A COMPLETE STRING CONSTANT COULD BE FOUND. A CLOSING DOUBLE-QUOTE MUST BE APPENDED TO COMPLETE THE STRING. STRING CONSTANTS CONSIST OF A DOUBLE-QUOTE, FOLLOWED BY A SERIES OF CHARACTERS, FOLLOWED BY A DOUBLE-QUOTE. IF A DOUBLE-QUOTE IS TO APPEAR IN THE STRING ITSELF, IT MUST BE CODED AS CONSECUTIVE DOUBLE-QUOTES ("").

BA199 **MG#FRNCD**
A FUNCTION REFERENCE MAY NOT BE CODED HERE. THE EXPRESSION REQUIRED IN THIS INSTANCE MUST BE A SIMPLE OR SUBSCRIPTED VARIABLE NAME. NO COMPUTATION IS PERMITTED IN THIS TYPE OF EXPRESSION.

BA200 **MG#LPNCD**
A LEFT-PAREN MAY NOT BE CODED HERE. THE EXPRESSION REQUIRED IN THIS INSTANCE MUST BE A SIMPLE VARIABLE OR A SUBSCRIPTED VARIABLE NAME. NO COMPUTATION IS PERMITTED IN THIS TYPE OF EXPRESSION.

BA201 **MG#FUNNR**
THIS FUNCTION DOES NOT REQUIRE ANY PARAMETERS BE PASSED TO IT. A LEFT-PAREN IS INVALID HERE. REMOVE THE PARAMETER LIST AND COMPLETE THE STATEMENT.

BA202 **MG#TWOSB**
MORE THAN TWO SUBSCRIPTS HAVE BEEN FOUND IN AN ARRAY REFERENCE. VECTORS MAY ONLY HAVE ONE SUBSCRIPT, AND ARRAYS MAY HAVE TWO. TO CORRECT THE ERROR, COMPLETE THE SUBSCRIPT BY ENTERING A RIGHT-PAREN, THEN CONTINUE THE STATEMENT.

BA203 **MG#TFCRQ**
THIS FUNCTION REQUIRES THAT PARAMETERS BE PASSED TO IT. TO PASS PARAMETERS, ENTER A LEFT-PAREN, THE LIST OF PARAMETER EXPRESSIONS SEPARATED BY A COMMA, AND A RIGHT-PAREN. FOR EXAMPLE: SIN(A—B),SEG$(A$,2,7).

HELP Command Message for Syntax Error

BA204 **MG#FCADR**
THIS FUNCTION REQUIRES ADDITIONAL PARAMETERS BE PASSED TO IT. ENTER THE ADDITIONAL PARAMETERS, THE RIGHT-PAREN, THEN CONTINUE THE STATEMENT.

BA205 **MG#OPNAL**
*AN OPERATOR (+, —, *, /, **, &) MAY NOT BE CODED HERE. THE EXPRESSION REQUIRED IN THIS INSTANCE MUST BE A SIMPLE OR SUBSCRIPTED VARIABLE NAME. NO COMPUTATION IS PERMITTED IN THIS TYPE OF EXPRESSION.*

BA206 **MG#UNMPA**
UNMATCHED PARENTHESES HAVE BEEN FOUND IN THIS EXPRESSION. THE ENTIRE EXPRESSION WILL NEED TO BE SCANNED, AND THE POSITION AND NUMBER OF PARENS ADJUSTED ACCORDINGLY.

BA207 **MG#EBCOP**
"EBC" OPERAND INVALID. VALID ARGUMENTS FOR THE EBC FUNCTION ARE:

ACK	*BEL*	*BS*	*CAN*	*CR*	*DC1*	*DC2*	*DC3*	*DC4*
DEL	*DLE*	*DS*	*EM*	*ENQ*	*EOT*	*ESC*	*ETB*	*ETX*
FF	*FS*	*GS*	*HT*	*LF*	*NAK*	*NUL*	*RS*	*SI*
SO	*SOH*	*SOS*	*SP*	*STX*	*SUB*	*SYN*	*US*	*VT*

OR ANY SINGLE CHARACTER.

BA208 **MG#UARPL**
*A UNARY PLUS SYMBOL IS NOT ALLOWED AT THIS POINT. AT THIS POINT IN THE EXPRESSION, OPERATORS (+, —, *, /, **), LEFT-PARENS, OR FUNCTION REFERENCED ARE NOT PERMITTED. A VARIABLE NAME OR CONSTANT MAY BE CODED HERE.*

Index

Term	Reference	Page
A		
ABS function	2.6	2—8
	6.5	6—7
	6.5.1	6—8
Addition operator	6.2	6—1
Ampersand operator	2.2	2—1
Argument list	2.6	2—8
Arithmetic expression	2.5	2—6
Arithmetic operations, hierarchy	6.2	6—1
Array variable	2.4	2—5
Assignment statement	3.4	3—10
Asterisk operator	2.2	2—1
ATN function	2.6	2—7
	6.5.1	6—7

Term	Reference	Page
B		
Batch processing		
background operation	8.3	8—3
BASIC in a batch environment	8.1	8—1
	8.2.2	8—1
	8.3	8—4
batch environment printout	Fig. 8—1	8—4
differences between batch mode and interactive mode	8.2	8—1
messages requiring a reply	8.2.1	8—1
RU command	8.2.4	8—2
syntax error messages	8.2.3	8—2
Built-in functions		
ABS	2.6	2—8
	6.5	6—7
	6.5.1	6—8
ATN	2.6	2—7
	6.5.1	6—7
CHR$	2.6	2—7
	6.5.3	6—12
CLK$	2.6	2—7
	6.5.3	6—12
COS	2.6	2—7
	6.5.1	6—7
COT	2.6	2—7
	6.5.1	6—7
DAT$	2.6	2—7
	6.5.3	6—12
description	6.5	6—7
DET	2.6	2—7
	6.5.2	6—10
EBC	2.6	2—7
	6.5.2	6—11

Term	Reference	Page
Built-in functions (cont)		
EXP	2.6	2—7
	6.5.1	6—7
file	6.5.2	6—8
INT	2.6	2—7
	6.5.2	6—8
LEN	2.6	2—7
	6.5.2	6—10
LOC	2.6	2—7
	6.5.4	6—13
LOF	2.6	2—7
	6.5.4	6—14
LOG	2.6	2—7
	6.5.1	6—8
mathematical	6.5.1	6—7
MOD	2.6	2—7
	6.5.2	6—10
NUM	2.6	2—7
	6.5.4	6—15
PER	2.6	2—7
	6.5.4	6—14
POS	2.6	2—7
	6.5.2	6—10
RND	2.6	2—7
	6.5.2	6—8
SEG$	2.6	2—7
	6.5.3	6—13
SGN	2.6	2—7
	6.5.2	6—9
SIN	2.6	2—7
	6.5.1	6—7
specialized	6.5.3	6—12
SQR	2.6	2—7
	6.5.1	6—8
STR$	2.6	2—7
	6.5.3	6—13
string	6.5.4	6—13
TAN	2.6	2—7
	6.5.1	6—7
TIM	2.6	2—7
	6.5.2	6—11
TYP	2.6	2—7
	6.5.4	6—15
URS$	2.6	2—7
	6.5.3	6—13
VAL	2.6	2—7
	6.5.2	6—11
BYE command	5.2.1	5—3
	Table A—1	A—2

C

Term	Reference	Page
CALL statement	3.8.1	3—60
	Table A—1	A—2
CHAIN statement	3.8.2	3—62
	Table A—1	A—2
CHANGE statement	3.9	3—68
	Table A—1	A—2
Channel setter	2.7	2—11
Characters		
delimiter	2.2	2—1
digit	2.2	2—1
letter	2.2	2—1
open-string character	2.2	2—2
special character	2.2	2—1
string character	2.2	2—2
CHR$ function	2.6	2—7
	6.5.3	6—12
CLK$ function	2.6	2—7
	6.5.3	6—12
Code, hints for efficient code	6.9	6—19
Command processor		
deleting program lines	1.6.7	1—8
file organization of a saved file	1.6.4	1—7
pause user program	1.6.8	1—8
program execution	1.6.1	1—5
program listing	1.6.2	1—6
returning control to the system	1.6.6	1—7
saving a program	1.6.3	1—7
terminating BASIC	1.6.9	1—8
using a saved program	1.6.5	1—7
Commands		
BYE	5.2.1	5—3
command format summary	Appendix A	
definitions	5.1.1	5—1
DELETE	5.2.2	5—4
HELP	5.2.3	5—5
introduction	5.1	5—1
LIST	5.2.4	5—6
MERGE	5.2.5	5—7
MODIFY	5.2.6	5—8
NEW	5.2.7	5—9
OLD	5.2.8	5—10
PRINT	5.2.9	5—11
RESEQUENCE	5.2.10	5—12
RUN	5.2.11	5—13
RUNOLD	5.2.12	5—14
SAVE	5.2.13	5—15
SYSTEM	5.2.14	5—16

Term	Reference	Page
Constants		
decimal numbers	2.3	2—2
definition	2.3	2—2
line numbers	2.3	2—3
string constants	2.3	2—3
Control statements		
END	3.5.1	3—12
FOR	3.5.2	3—13
GOSUB	3.5.3	3—16
GOTO	3.5.4	3—17
IF	3.5.5	3—18
NEXT	3.5.2	3—13
ON	3.5.6	3—20
PAUSE	3.5.7	3—21
RANDOMIZE	3.5.9	3—23
RETURN	3.5.3	3—16
STOP	3.5.8	3—22
SYSTEM	3.5.11	3—25
TIME	3.5.10	3—24
COS function	2.6	2—7
	6.5.1	6—7
COT function	2.6	2—7
	6.5.1	6—7

D

Term	Reference	Page
Data input/output statements		
DATA	3.6.5	3—33
INPUT	3.6.1	3—26
LINPUT	3.6.2	3—27
MARGIN	3.6.3	3—28
PRINT	3.6.4	3—29
READ	3.6.5	3—33
RESET	3.6.6	3—35
RESTORE	3.6.6	3—35
USING	3.6.7	3—36
DATA statement	3.6.5	3—33
	Table A—1	A—2
DAT$ function	2.6	2—7
	6.5.3	6—12
Debugging	7.1	7—1
Declaration statements		
DEF statement	3.2.1	3—4
DIM statement	3.2.2	3—6
FNEND statement	3.2.3	3—8
DEF statement	3.2.1	3—4
	Table A—1	A—2
DELETE command	5.2.2	5—4
	Table A—1	A—2
Deletion		
DELETE command	5.2.2	5—4
deleting program lines	1.6.7	1—8
Delimiter		
operator	2.2	2—1
separator	2.2	2—1
DET function	2.6	2—7
	6.5.2	6—10
Digit	2.2	2—1
DIM statement	3.2.2	3—6
	Table A—1	A—2
Division operator	6.1	6—1

Term	Reference	Page
E		
EBC function	2.6	2—7
	6.5.2	6—11
END statement	3.5.1	3—12
	Table A—1	A—2
Error messages	Appendix C	
Errors		
description	7.1	7—1
error messages	Appendix C	
logic	7.2	7—1
preventing running of program	7.3	7—2
Execution, program	1.6.1	1—5
EXP function	2.6	2—7
	6.5.1	6—7
Exponentiation operator	6.1	6—1
Expressions		
arithmetic expression	2.5	2—6
description	2.5	2—6
string expression	2.5	2—6

Term	Reference	Page
F		
Factor	2.5	2—6
File description	4.2	4—1
File functions		
description	2.6	2—7
LOC	6.5.4	6—13
LOF	6.5.4	6—14
MAR	6.5.4	6—13
NUM	6.5.4	6—15
PER	6.5.4	6—14
TYP	6.5.4	6—15
File INPUT statement	4.3.2	4—10
	Table A—1	A—3
File LINPUT statement	4.3.3	4—12
File MARGIN statement	4.3.4	4—13
	Table A—1	A—3
File MAT INPUT statement	4.3.5	4—14
	Table A—1	A—4
File MAT LINPUT statement	4.3.5	4—14
	Table A—1	A—4
File MAT PRINT statement	4.3.5	4—14
	Table A—1	A—4
File MAT READ statement	4.3.5	4—14
	Table A—1	A—4
File MAT WRITE statement	4.3.5	4—14
	Table A—1	A—4
File name	5.1.1	5—1
File PRINT statement	4.3.6	4—16
	Table A—1	A—5
File READ statement	4.3.7	4—18
	Table A—1	A—5
File RENAME statement	4.3.8	4—19
File RESET statement	4.3.9	4—20
	Table A—1	A—5

Term	Reference	Page
File SCRATCH statement	4.3.10	4—21
	Table A—1	A—5
FILE statement	4.3.1	4—5
	Table A—1	A—2
File statements		
FILE	4.3.1	4—5
general	4.3	4—3
INPUT	4.3.2	4—10
LINPUT	4.3.3	4—12
MARGIN	4.3.4	4—13
matrix I/O statements	4.3.5	4—14
overview	Table 4—1	4—3
PRINT	4.3.6	4—16
READ	4.3.7	4—18
RENAME	4.3.8	4—19
RESET	4.3.9	4—20
SCRATCH	4.3.10	4—21
WRITE	4.3.11	4—22
File support	4.1	4—1
File WRITE statement	4.3.11	4—22
	Table A—1	A—6
Files		
description	4.2	4—1
library files	4.2	4—1
MIRAM files	4.2	4—1
statements	4.3	4—3
support	4.1	4—1
temporary	4.2	4—1
use	6.8	6—17
FNEND statement	3.2.3	3—8
	Table A—1	A—2
FOR statement	3.5.2	3—13
	Table A—1	A—2
Formats		
commands	Table A—1	A—2
statements	Table A—1	A—2
summary of formats	Appendix A	
Formatting output		
numeric output	3.6.7.2	3—38
string output	3.6.7.1	3—37
Functions		
built-in	6.5	6—7
file	6.5.4	6—13
mathematical	6.5.1	6—7
multiline	6.6	6—15
references	2.6	2—7
specialized	6.5.2	6—8
string	6.5.3	6—12

Term	Reference	Page
G		
GOSUB statement	3.5.3	3—16
	Table A—1	A—2
GOTO statement	3.5.4	3—17
	Table A—1	A—2
H		
HELP command	5.2.3	5—5
I		
IF statement	3.5.5	3—18
	Table A—1	A—3
INPUT statement	3.6.1	3—26
	Table A—1	A—3
INT function	2.6	2—7
	6.5.2	6—8

Term	Reference	Page
L		
Language elements		
channel setter	2.7	2—11
characters	2.2	2—1
constants	2.3	2—1
description	2.1	2—1
expressions	2.5	2—6
function references	2.6	2—7
statements	2.8	2—12
variables	2.4	2—4
LEN function	2.6	2—7
	6.5.2	6—10
LET statement	3.4	3—10
	Table A—1	A—3
Letter	2.2	2—1
Library files	4.2	4—1
LIBRARY statement	3.8.3	3—63
	Table A—1	A—3
Line number	1.4	1—3
	2.3	2—3
	5.1.1	5—1
LINPUT statement	3.6.2	3—27
LIST command	5.2.4	5—6
	Table A—1	A—3
Lists, use	6.4	6—5
LOC function	2.6	2—7
	6.5.4	6—13
LOF function	2.6	2—7
	6.5.4	6—14
LOG function	2.6	2—7
	6.5.1	6—8
Logic errors	7.2	7—1
LOGOFF procedure	1.7	1—8
LOGON procedure	1.3	1—3
Loops		
nested loops	Table 6—1	6—5
use	6.3	6—3

Term	Reference	Page
M		
MARGIN statement	3.6.3	3—28
	Table A—1	A—3
MAT addition statement	3.7.2	3—45
MAT constant statement	3.7.3	3—48
MAT identity statement	3.7.4	3—49
MAT INPUT statement	3.7.5	3—50
	Table A—1	A—4
MAT inversion statement	3.7.6	3—51
MAT LINPUT statement	3.7.7	3—52
	Table A—1	A—4
MAT multiplication statements	3.7.2	3—45
MAT null statement	3.7.8	3—52
MAT PRINT statement	3.7.9	3—54
	Table A—1	A—4
MAT READ statement	3.7.10	3—55
	Table A—1	A—4
MAT scalar multiply statement	3.7.11	3—56
MAT statement	3.7	3—42
	Table A—1	A—3
MAT subtraction statement	3.7.2	3—45
MAT transpose statement	3.7.12	3—57
MAT vector multiplication statement	3.7.13	3—58
MAT zeros statement	3.7.14	3—59
Mathematical functions		
ABS	6.5.1	6—8
ATN	6.5.1	6—7
COS	6.5.1	6—7
COT	6.5.1	6—7
description	2.6	2—7
EXP	6.5.1	6—7
LOG	6.5.1	6—8
SIN	6.5.1	6—7
SQR	6.5.1	6—8
TAN	6.5.1	6—7
Matrix dimensions	3.7.1	3—44

Term	Reference	Page
Matrix I/O statements	4.3.5	4—13
Matrix operation statements		
MAT addition, subtraction, and multiplication	3.7.2	3—45
MAT constant	3.7.3	3—48
MAT identity	3.7.4	3—49
MAT INPUT	3.7.5	3—50
MAT inversion	3.7.6	3—51
MAT LINPUT	3.7.7	3—52
MAT null	3.7.8	3—53
MAT PRINT	3.7.9	3—54
MAT READ	3.7.10	3—55
MAT scalar multiply	3.7.11	3—56
MAT transpose	3.7.12	3—57
MAT vector multiplication	3.7.13	3—58
MAT zeros (0's)	3.7.14	3—59
Matrix dimensioning	3.7.1	3—44
MERGE command	5.2.5	5—7
	Table A—1	A—4
MIRAM files	4.2	4—1
Mnemonics	Table 2—1	2—10
MOD function	2.6	2—7
	6.5.2	6—10
MODIFY command	5.2.6	5—8
Multiline functions, use	6.6	6—15
Multiplication operator	6.1	6—1

N

Term	Reference	Page
Nested loops	Table 6—1	6—5
NEW command	5.2.7	5—9
	Table A—1	A—4
NEXT statement	3.5.2	3—13
	Table A—1	A—4
Numeric array	2.4	2—5
Numeric variable	2.4	2—4
NUM function	2.6	2—7
	6.5.4	6—15

O

Term	Reference	Page
OLD command	5.2.8	5—10
	Table A—1	A—4
ON statement	3.5.6	3—20
	Table A—1	A—4
Open-string character	2.2	2—2
Output, formatting		
numeric output	3.6.7.2	3—38
string output	3.6.7.1	3—37

Term	Reference	Page
P		
Password	5.1.1	5—2
PAUSE statement	3.5.7 Table A—1	3—21 A—5
Pause user program	1.6.8	1—8
PER function	2.6 6.5.4	2—7 6—14
POS function	2.6 6.5.2	2—7 6—10
Primary	2.5	2—6
PRINT command	5.2.9 Table A—1	5—11 A—5
PRINT statement	3.6.4 Table A—1	3—29 A—5
Processor	See command processor.	
Program segmentation		
CALL statement	3.8.1	3—60
CHAIN statement	3.8.2	3—62
introduction	3.8	3—59
LIBRARY statement	3.8.3	3—63
SUB statement	3.8.4	3—64
SUBEND statement	3.8.5	3—66
SUBEXIT statement	3.8.6	3—67
Program techniques	6.1	6—1
Programs		
deleting program lines	1.6.7	1—8
execution	1.6.1	1—5
listing	1.6.2	1—6
pausing program execution	1.6.8	1—8
saving a program	1.6.3	1—7
terminating BASIC	1.6.9	1—8
using a saved program	1.6.5	1—7

Term	Reference	Page
R		
RANDOMIZE statement	3.5.9 Table A—1	3—23 A—5
READ statement	3.6.5 Table A—1	3—33 A—5
Relation symbols	Table 3—2	3—18
REM statement	3.3 Table A—1	3—9 A—5
Remark statement	3.3	3—9
RESEQUENCE command	5.2.10 Table A—1	5—12 A—5
RESET statement	3.6.6 Table A—1	3—35 A—5
RESTORE statement	3.6.6	3—35
RETURN statement	3.5.3 Table A—1	3—16 A—5
RND function	2.6 6.5.2	2—7 6—8
RU command	8.2.4	8—2
RUN command	5.2.11 Table A—1	5—13 A—5
RUNOLD command	5.2.12 Table A—1	5—14 A—5

Term	Reference	Page
S		
Sample session	Appendix B	
	Fig. B—1	B—1
SAVE command	5.2.13	5—15
	Table A—1	A—5
Saving a program		
file organization	1.6.4	1—7
SAVE command	1.6.3	1—7
	5.2.13	5—15
use	1.6.5	1—7
Scalar variable	2.4	2—4
SEG$ function	2.6	2—7
	6.5.3	6—13
SGN function	2.6	2—7
	6.5.2	6—9
SIN function	2.6	2—7
	6.5.1	6—7
Source program construction	1.4	1—3
Special characters	2.2	2—1
Specialized functions		
description	2.6	2—7
DET	6.5.2	6—10
EBC	6.5.2	6—11
INT	6.5.2	6—8
LEN	6.5.2	6—10
MOD	6.5.2	6—10
POS	6.5.2	6—10
RND	6.5.2	6—8
SGN	6.5.2	6—9
TIM	6.5.2	6—11
VAL	6.5.2	6—11
SQR function	2.6	2—7
	6.5.1	6—8
Statements		
assignment	3.4	3—10
change	3.9	3—68
control	3.5	3—11
declaration	3.2	3—3
description	2.8	2—12
executable	2.8	2—12
	Table 3—1	3—1
I/O	3.6	3—25
matrix operation	3.7	3—42
nonexecutable	2.8	2—12
	Table 3—1	3—1
program segmentation	3.8	3—59
remark	3.3	3—9
summary of formats	Appendix A	
STOP statement	3.5.8	3—22
	Table A—1	A—5
String		
array variable	2.4	2—5
built-in functions	See string built-in functions.	
character	2.2	2—2
constant	2.3	2—3
expression	2.5	2—6
primary expression	2.5	2—7
variable	2.4	2—4
String built-in functions		
CHR$	2.6	2—7
	6.5.3	6—12
CLK$	2.6	2—7
	6.5.3	6—12
DAT$	2.6	2—7
	6.5.3	6—12
SEG$	2.6	2—7
	6.5.3	6—13
STR$	2.6	2—7
	6.5.3	6—13
URS$	2.6	2—7
	6.5.3	6—13
STR$ function	2.6	2—7
	6.5.3	6—13
SUB statement	3.8.4	3—64
	Table A—1	A—5
SUBEND statement	3.8.5	3—66
	Table A—1	A—6
SUBEXIT statement	3.8.6	3—67
	Table A—1	A—6
Subprograms	6.7	6—16
Subtraction operator	6.1	6—1
Syntax checker	1.5	1—4
SYSTEM command	5.2.14	5—16
	Table A—1	A—6
System overview	1.1	1—1
	Fig. 1—1	1—2
SYSTEM statement	3.5.11	3—25
	Table A—1	A—6

Term	Reference	Page
T		
Tables, use	6.4	6—5
TAN function	2.6 6.5.1	2—7 6—7
Temporary files	4.2	4—1
Term	2.5	2—6
Terminals supported by BASIC	1.2	1—2
Terminating BASIC		
BYE command	1.6.9 5.2.1	1—8 5—3
LOGOFF procedure	1.7	1—8
TIM function	2.6 6.5.2	2—7 6—11
TIME statement	3.5.10 Table A—1	3—24 A—6
TYP function	2.6 6.5.4	2—7 6—15
U		
URS$ function	2.6 6.5.3	2—7 6—13
User-defined function	2.6	2—8
USING statement		
description	3.6.7 Table A—1	3—36 A—6
formatting numeric output	3.6.7.2	3—38
formatting string output	3.6.7.1	3—37
use with PRINT statement	3.6.7.3	3—39
V		
VAL function	2.6 6.5.2	2—7 6—11
Variables		
array	2.4	2—5
scalar	2.4	2—4
Volume	5.1.1	5—2

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

_____

(Document Title)

_____ _____ _____

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

Cut along line.

From:

_____

(Name of User)

_____

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

CUT

FOLD

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

_____

(Document Title)

_____ _____ _____

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

Cut along line.

From:

_____

(Name of User)

_____

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

CUT

FOLD

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

_____

(Document Title)

_____ _____ _____

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

Cut along line.

From:

_____

(Name of User)

_____

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

FOLD