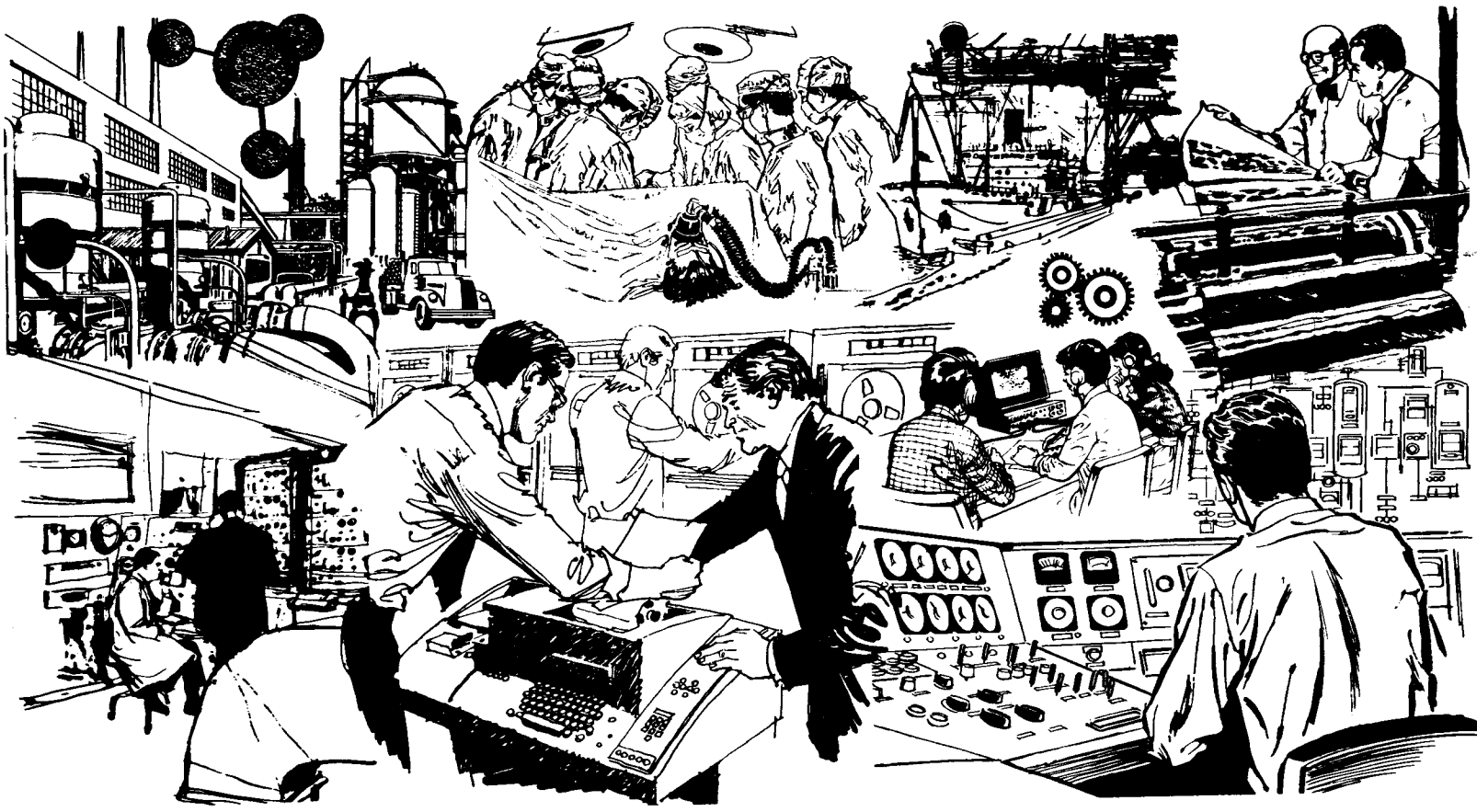
BEM:BASIC — OS/3

User Reference



SPERRY UNIVAC

UA-0140 Rev. 3

SPERRY UNIVAC

communique

FROM APPLICATIONS SOFTWARE

DOCUMENT UPDATE MEMORANDUM

December 1980

TITLE: BEM:BASIC

User Reference

DOCUMENT NUMBER: UA-0140 Rev. 3 — Update C

OPERATING SYSTEM: OS/3

ABSTRACT: This update package incorporates all changes made necessary by the latest release of this program product.

Proper insertion of these pages into your current UA-0140 Rev. 3 manual will provide an accurate description of the latest release version of this applications software product.

NOTE: The application programs described in this document are confidential information and proprietary products of the Sperry Univac Division.

SPERRY UNIVAC

communique

FROM APPLICATIONS SOFTWARE

DOCUMENTATION UPDATE MEMORANDUM

July 1978

TITLE: BEM:BASIC — OS/3
User Reference

DOCUMENT NUMBER: UA-0140 Rev. 3 — Update A

ABSTRACT: This announces the release and availability of Update Package A for the *BEM:BASIC User Reference,* UA-0140.

This update package includes minor changes in the text, as well as documentation of new features such as RESEQUENCE, MERGE, and SYSTEM. Changes made for Level 5.0 BEM are also documented.

NOTE: The BEM:BASIC application programs described in this document are confidential information and proprietary products of the Sperry Univac Division.

ORDERING PROCEDURE: The update package alone, or the manual plus the update package may be requisitioned. To receive **only** the update, order UA-0140 Rev. 3 Update A; to receive both the manual and the update, order the current manual and the update package.

Domestic offices order via Sales Help Requisition (form UD1-578) from Customer Information Distribution Center (CIDC), 555 Henderson Road, King of Prussia, Pa. 19406. Locations outside U.S.A. order from International Distribution in the usual manner.

PAGE STATUS SUMMARY

Update C to UA-0140 Rev. 3

Section	Page Number	Update Level
Cover		Orig.
Title		Orig.
Copyright		C
Preface	*iii-iv*	Orig.
Contents	*v-ix*	A
1	1	A
	2	Orig.
	3-4	A
	5-9	Orig.
2	1-9	Orig.
	10-11	A
3	1	Orig.
	2	A
	3-16	Orig.
	16A	A
	17-20	Orig.
	21	B
	22	A
	23	B
	24	Orig.
	25	A
	26	Orig.
	27	A
	28-29	Orig.
	30-31	A
	32-47	Orig.
4	1-6	Orig.
	7	C
	8	C
	9-19	Orig.
5	1	Orig.
	2	B
	3	Orig.
	4	A
	5	Orig.
	6-7	A
6	1-11	Orig.
	12	B
	13	Orig.
	14-16	B
	17	Orig.
	18	A
	19-21	Orig.
7	1	A
	2-3	Orig.
8	1-16	A

Section	Page Number	Update Level
A	1	Orig.
	2	C
	3,4	Orig.
	5-7	A
	8	Orig.
B	1-2	B
C	1-27	A

Section	Page Number	Update Level

All technical changes are denoted by an arrow (⟶) in the margin. A downward arrow (↓) next to a line indicates that technical changes begin at this line and continue until an upward 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.

BEM:BASIC — OS/3

User Reference

SPERRY UNIVAC

This document contains the latest information available at the time of publication. However, Sperry Univac reserves the right to modify or revise its contents. To ensure that you have the most recent information, contact your local Sperry Univac representative.

SPERRY UNIVAC NEWSCOMP Newspaper Composition Program was used by Application Services in typesetting this publication.

A User Comment Sheet is provided at the back of this publication for your comments. If the sheet has been removed, comments may be mailed to Sperry Univac, Attn: Manager, Application Services, P.O. Box 500, Blue Bell, PA. 19424.

Sperry Univac is a division of Sperry Corporation.

Printed in U.S.A.

PREFACE

This reference manual describes the BASIC (Beginner's All-Purpose Symbolic Instruction Code) system which permits the user to prepare, test, and execute programs while operating from a remote terminal. This version of BASIC operates under the SPERRY UNIVAC Basic Editor Monitor (BEM) of the OS/3 Operating System.

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.

System 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 that is available to the user in constructing the BASIC program.

Section 4 — BASIC File Support — describes the file-related statements and access methods supported under BASIC.

Section 5 — BASIC Commands — describes each BASIC edit command that is available in preparing BASIC programs. These commands allow the user to name a program, execute a program, manipulate the source language statements in a program, and return control to BEM.

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, matrices, built-in functions, and multiline functions.

Section 7 — Errors and Debugging — describes the various user errors which may occur in preparing a BASIC program and the required correction facilities.

Section 8 — BEM Operation — details elements of the BEM monitor, and how to use it.

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 — System Error Messages — documents all of the BASIC and BEM error messages.

Although BASIC is a self-contained language system requiring minimal interaction between the control program and the user, it is nevertheless advisable that the user become acquainted with the information contained in the following publications:

BEM — OS/3 Basic Editor Monitor, User Reference UA-0139
BEM:EDT — OS/3 Interactive Editor, User Reference UA-0141
BEM:RSP — OS/3 Remote Spoolout Processor, User Reference UA-0243

CONTENTS

3 SOURCE LANGUAGE STATEMENTS

6 BASIC PROGRAM TECHNIQUES

7 ERRORS AND DEBUGGING

8 BEM OPERATION

APPENDIXES

1 SYSTEM DESCRIPTION

1.1 GENERAL

The SPERRY UNIVAC Basic Editor Monitor (BEM) system provides the remote terminal user with the capability of generating, modifying, and executing programs written in Beginner's All-Purpose Symbolic Instruction Code (BASIC). The BASIC system also provides the user with the capability of saving the programs in OS/3 Library files for subsequent processing and updating.

Figure 1-1 shows an overview of the BASIC system. After logging on, the BASIC system is invoked by typing the /EXEC BASIC command at the remote terminal. The system then loads the BASIC compiler, responds with READY, and the user begins to construct or modify his source program. Each BASIC statement that is entered is immediately analyzed by a Syntax Checker for syntax errors such as invalid constants, expressions, and construction. If an error is detected, BASIC types 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 instruct the BASIC system 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. (Refer to 1.6.1, "Program Execution.") If an error is detected, BASIC returns the line number of the source statement in error and appropriate diagnostic message to the user. (Refer to Section 7, "Errors and Debugging" and Appendix C, System Error Messages.)

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.

1.2 TERMINALS SUPPORTED BY BASIC

The BASIC system supports all terminals supported by BEM.

Lines of source text and editing commands must be constructed using only the minimum UNISCOPE character set with the following exceptions for teletypewriters:

1. A vertical arrow (↑) is accepted as a substitute for **, which denotes exponentiation.

2. The designation TRANSMIT is used to show the UNISCOPE terminal TRANSMIT KEY. This is equivalent to the ETX key on a DCT terminal (CTRL "C").

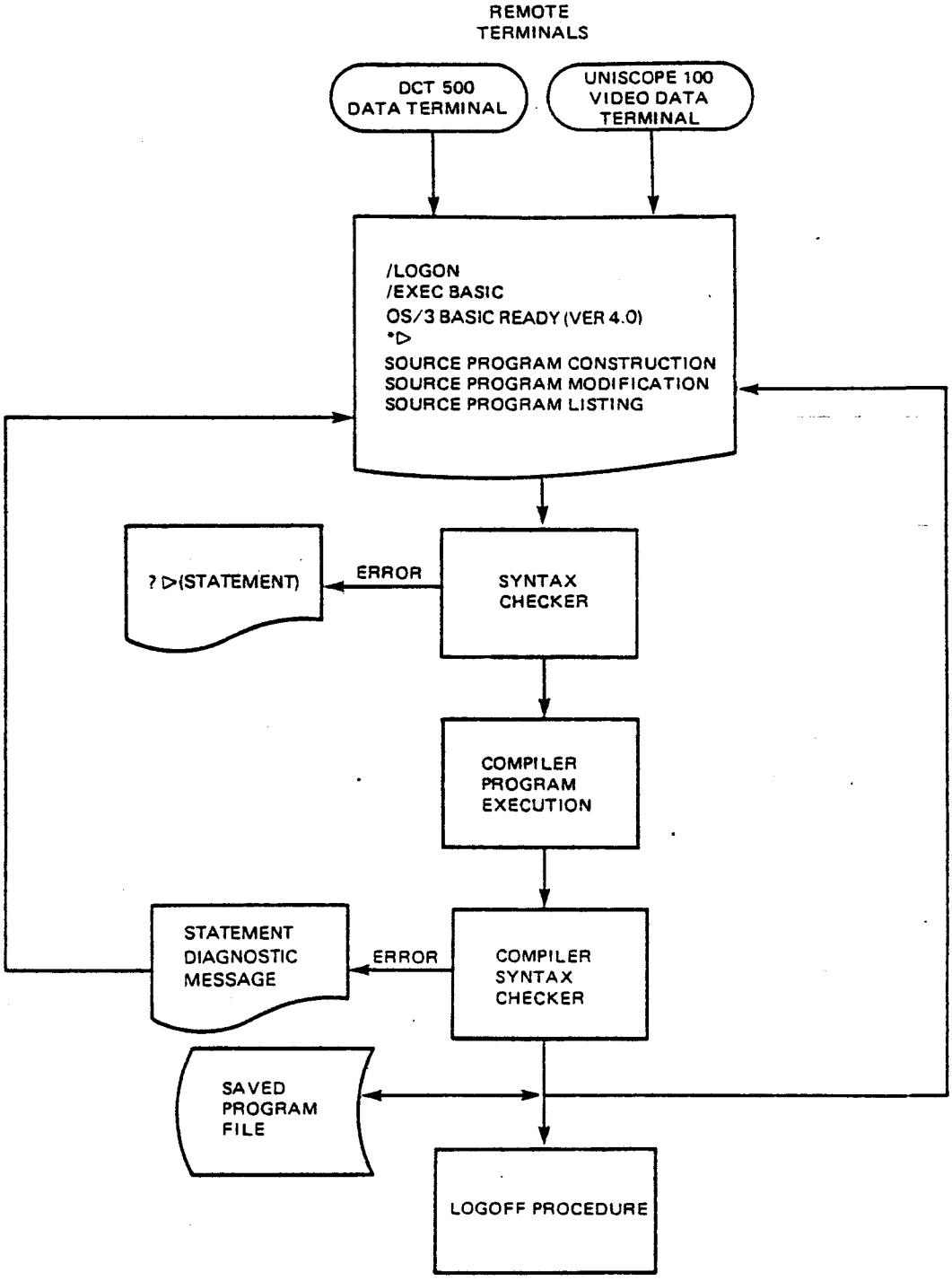


Figure 1-1 BASIC System Overview

1.3 LOGON PROCEDURE

In order to be initially connected to the operating system, the user must enter the /LOGON command from his terminal. This command identifies the user to the operating system and initiates the user task. The format of the LOGON command is:

/LOGON *userid, account, password*

where

LOGON	specifies that the user wants to log on and initiate a task.
userid	specifies the user identification (one to four alphanumeric characters).
account	is the optional account id for this user (one to four alphanumeric characters).
password	is the optional user-id/account-id protection password (one to four alphanumeric characters).

For a more detailed discussion of the LOGON command, refer to the *BEM — OS/3 Basic Editor Monitor User Reference* UA-0139.

1.4 SOURCE PROGRAM CONSTRUCTION

The user invokes BASIC by issuing the following Executive command:

/EXEC BASIC

Control is transferred to BASIC which immediately responds

```
OS/3 BASIC READY (VER 5.0) 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 invoked, 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. The BASIC source language statements and editing commands are described in detail in Sections 3, 4, and 5 respectively. 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 one to five digits of 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 work space, is not saved unless the user issues a SAVE 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 128 characters including any backspaces, carriage returns, or other non-BASIC characters. However, only the first 80 columns will be scanned for source lines.

Since 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 TRANSMIT deletes a statement with matching line number in the current user's program file.

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 TRANSMIT as the next line of input.

Example:

The user types in the line.

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

The system responds

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

since the GOTO following THEN is incorrect.

The user may then type in

```
41  TRANSMIT
```

and the complete statement

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

is processed by BASIC.

The following types of errors are detected by the BASIC Syntax Checker:

- Incorrect constants, identifiers, functions 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 re-transmit it to BASIC. In the case where the user does not wish to correct the statement, but wishes to enter a new statement or a command, the following action should be taken:

- On a UNISCOPE terminal, back up the cursor to the Start-of-Entry symbol (▷) and erase the line. A new statement may now be entered.
- On a hardcopy terminal, transmit a percent sign (%). BASIC will respond with an asterisk (*) indicating a new statement may 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 which 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 which caused the error and an appropriate diagnostic.

Example:

```
INVALID NESTING OF FOR-NEXT STATEMENTS
LOADER 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 indices
- Statements leading to unpredictable results
- Object code exceeding available memory
- 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:

```
UNCORRECTED ERROR IN SOURCE PROGRAM
LOADER AT LINE 00766
```

The user should go back and correct the line(s) 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 typeout to the user's terminal consisting of the source statement which resulted in the error and an appropriate diagnostic.

Example:

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

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

1.6.2 Program Listing

The LIST or PRINT command can be used to display all or parts 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 a SAT 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:

```
OVERWRITE PREVIOUS FILE (YES, NO)?
```

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 direct-access file space into his work space. 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 work space.

1.6.6 Returning Control to the Monitor

During the BASIC session, it may be necessary for the user to return control to the monitor, so that certain monitor commands such as STATUS, TYPE, etc., may be issued. In order to facilitate returning control to the monitor, BASIC provides the user with the SYSTEM command. The SYSTEM command causes BASIC to interrupt to the monitor, and the user can subsequently return to BASIC by issuing the /R[ESUME] command.

1.6.7 Deleting Program Lines

BASIC statements which 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 Terminating BASIC

When the user has finished with BASIC the BYE command may be used to terminate BASIC and return any storage space occupied by the program.

1.7 LOGOFF PROCEDURE

The monitor LOGOFF command terminates the user session. Its purpose is to end the task and return to the BEM system the memory and any disk space used by the task.

This command must be the last one in the task and is entered according to the following format:

Format:

```
/LOGOFF
```

2 LANGUAGE ELEMENTS

2.1 GENERAL

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

- Characters
- Constants
- Variables
- Expressions
- Function references
- 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:	, . ; Δ ":
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 (").
string character	letter, digit, operator, special character, comma (,), period (.), a semicolon (;), or a blank (Δ).

NOTES:

1. *blanks:*

   The character blank, which may be used in constructing the BASIC programs, is designated in the syntax by the symbol Δ. Any spaces which 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 in 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, two consecutive quotes are used.

3. *asterisk:*

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

2.3 CONSTANTS

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

decimal numbers

A fraction which may be optionally followed by an exponent field. A fraction is defined to be a series of one or more digits and 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 ten by which the fraction is to be multiplied and consists of the letter E followed by an optional sign and one or two 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 constants

closed string: 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

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

NOTES

1. *decimal numbers:*

   All decimal numbers are converted and stored internally in floating-point format. The exponent occupies seven 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

   is acceptable, but is (effectively) rounded to

   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. Within a closed-string constant, two consecutive quotes are interpreted as a single quote. Note that is is not possible to enter a string constant in a program longer than 74 characters, since the maximum line length is 80 characters.

3. *line numbers:*

   Each statement in a BASIC program must be preceeded by a line number which is an integer between 1 and 99999. The line numbers specify the logical sequence of statements in a program (increasing 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 000175 is equivalent to 175.

2.4 VARIABLES

Variables are used in a BASIC program 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	numeric variable or string variable
numeric variable	letter optionally followed by a single digit Examples: X, X2
string variable	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	numeric array variable or string array variable
numeric array variable	letter followed by one or two subscript expressions enclosed in parentheses Examples: X(4), X(4,20), X(A+B)
string array variable	letter followed by a dollar sign ($) followed by one or two subscripts enclosed in parentheses Examples: C$ (20), C$(A+B), D$ (A,C)

NOTES:

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 means of a dimension (DIM) statement. (See Section 3). An implicit upper bound of 10 for either dimension is implied if not specified. In either case, the lower bound is always zero (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 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). The maximum number of characters which may be placed in a string variable is 4095.

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 are used primarily to identify input/output. Unless otherwise stated, all expressions are assumed to designate single values.

arithmetic expression	term optionally preceded by a minus (−) or plus (+) sign; or an arithmetic expression plus (+) or minus (−) a term Example A**2*B-3
term	factor or a term multiplied (*) or divided (/) by a factor Example A**2*B
factor	primary or a factor raised to a power (**) designated by a primary Example: A**2

primary	decimal number, numeric reference, function reference, or an arithmetic expression which is enclosed in parentheses Examples: 2,A,RND(X), (C—D)
string expression	string primary or a string expression followed by an ampersand (&) denoting concatenation, followed by another string expression. Example: "ABC" & B$
string primary	closed string, string reference, or function reference Examples: A$, SEG$(D$, 6, 8), "AB"

NOTES:

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 only be raised 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 either specific built-in functions provided within the BASIC system, or user-defined functions. All function references consist of a function name followed by an argument list enclosed within parentheses. All built-in functions have between zero and three arguments. In each case, the arguments are evaluated and control is transferred to an out-of-line routine for evaluating the referenced function. The resulting (single) 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 EXP 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". . .

NOTES:

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

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

3. *LOG(x) designates the natural logarithm of x, ln 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 pseudo random number:*

   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 pseudo random 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).*

   d. *If no argument is used, x = 0 is assumed. To generate a sequence of pseudo random 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.*

   For 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 pseudo-function 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 which 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.

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 4- 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.*

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 Data Management 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.*

28. *NUM returns the number of values inputted for the last vector MAT INPUT statement.*

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

2.7 CHANNEL SETTER

The channel setter is used in file-related statements to specify which data file is to be selected. It has the form:

 # expression

where

#	identifies the channel setter.
expression	is a numeric expression which is evaluated at execution time.

Examples: #3, #I, #3-J

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.

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	line number followed by an executable statement or a nonexecutable statement
executable statement	assign, control, input-output, matrix, or 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 either classified as executable or nonexecutable. The executable statements are categorized as assignment, change of control, input/output, matrix operations, and data files. The nonexecutable statements are categorized as declarative and remark. Table 3-1 shows the list of all the BASIC source language statements.

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 memory available to load the program, and is not a limit imposed by the compiler.

In describing the statements, the following conventions are used:

- Keywords that may be used in the statement are shown in capital letters.
- Names constructed using lower case letters and embedded hyphens designate syntactic variables.
- Brackets, [ ], are used to enclose optional parameters.
- Braces, { }, are used to enclose alternatives.
- Ellipsis, . . . ,following an operand parameter indicates that the user may specify more than one parameter of that type. For example, the syntax

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

  allows the statements

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

  to contain many input variables in the READ list.

Table 3-1 List of BASIC Statements

Statement Category	Statements	
	Executable	Nonexecutable
Declaration		DIM DEF FNEND
Remark		REM
Assignment	LET	
Control	FOR and NEXT GOSUB and RETURN GOTO IF ON STOP, PAUSE, and END SYSTEM	TIME
Matrix Operations	MAT-add, subtract, multiply MAT-invert, transpose MAT-scalar multiply MAT-identity, constant, zero MAT-null MAT INPUT MAT LINPUT MAT READ MAT PRINT MAT WRITE	
Data File	READ RESTORE, RESET	DATA
Program Subdivision	CHAIN CALL SUBEXIT SUBEND	SUB
Conversion	CHANGE	
Input/Output and Files	FILE INPUT LINPUT MARGIN PRINT and USING READ RENAME RESET SCRATCH WRITE	

3.2 DECLARATION STATEMENTS

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

3.2.1 DIM Statement

The DIM statement explicitly specifies the upper bound(s) of numeric and string arrays so that sufficient space can be reserved in memory for the array. Either 1- or 2- dimensional numeric or string arrays can be dimensioned. The lower bound for each dimension is always 0.

Format

$$\text{DIM} \quad \left\{ \begin{matrix} \textit{numeric-dimension} \\ \textit{string-dimension} \end{matrix} \right\} \left[, \left\{ \begin{matrix} \textit{numeric-dimension} \\ \textit{string-dimension} \end{matrix} \right\} \ldots \right]$$

where

numeric-dimension — 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 — 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, as long as the number of subscripts remains consistent.
5. The upper limit on the subscripts of an array will be 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.

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(2), . . . ,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, 30), B(1, 30), . . . , B(20, 30)

and R$ is a 1-dimensional string array consisting of 36 string variables: R$(0), R$(1), R$(2), . . . , R$(35)

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 which can be referenced by subscripts or matrix operations.

3.2.2 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

FN*letter*	The name of the defined function 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 which defines the exit from the function. Branching into and out of a multiline function definition 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. In order 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 statement(s) defining the function are bypassed and control passes to the first 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 which are passed in subprogram CALLs must be defined prior to the CALL statement.

Example: 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:

```
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 and accept and answer by referencing the user function FNA$.

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

The REM statement provides a means for inserting explanatory remarks in 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
279   LET A7 = B +4 'THIS IS A COMMENT
280   LET P =4*ATN(1) 'COMPUTE THE VALUE OF "PI"
290   'THE FOLLOWING CODE USES P FOR PI.
```

3.4 ASSIGNMENT STATEMENT

The LET statement is used to assign 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	FNletter [$] = expression

Programming Notes

1. The statement verb LET need not be written.
2. Mixed mode assignment will not be 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 assignment.

4. The function-let is used to assign a value to a multiline user-defined function. (See DEF statement.)

Example 1:

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

when statement 20 is executed, I is assigned the value 3.5 and then A(3) is assigned the value 3.5.

Example 2:

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

This is a string-let statement used to assign 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: FOR and NEXT, GOSUB and RETURN, GOTO, IF, ON, STOP, and END statements.

3.5.1 FOR and NEXT Statements

The FOR statement initiates a loop and 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 $(f-i)*s \geq 0$. 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 FOR I = 1 TO 10 STEP 2
   ```

   is the same as

   ```
   10 FOR I = 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 (¼), 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)
```

3.5.2 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.

3.5.3 GOTO Statement

The GOTO statement causes an unconditional transfer of control to the statement whose line number is referenced.

Format

GO TO *line-number*

Programming Note

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

Example:

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

In this example, the GOTO statement (line-number 20) transfers control to the assignment statement at line number 25 and thereby bypasses the READ statement (line-number 21).

3.5.4 IF Statement

The IF statement provides a conditional transfer of control. If the condition specified is true, then control is transferred to the line number referenced.

Format

IF *condition* { THEN / GOTO / GOSUB } *line-number*

where

condition { arithmetic-expression relation arithmetic-expression / string-expression relation string-expression / END channel-setter / MORE channel-setter }

relation 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
<= or =<	Is less than or equal to	A <= B
>	Is greater than	A > B
>= or =>	Is greater than or equal to	A >= B
<> or ≠	Is not equal to	A <> B

Programming Notes

1. Mixed mode expression across a relation will not be 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 which an INPUT would accept.

Example 1:

```
10 A$="ASHLEY"
20 B$="BOB"
30 IF A$<B$ THEN 50
     .
     .
     .
     .
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 will transfer 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 will be transferred to the statement with line number 80.

Example 3:

```
175  IF MORE #7 THEN 190
180  PRINT "PROGRAM FINISHED"
185  STOP
190  READ #7:V$
```

This example uses the MORE condition to control the reading of a file. When the last record has been processed, the program stops.

3.5.5 ON Statement

The ON statement provides a multibranched switch.

Format

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

Programming Notes

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

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

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

Example:

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

The first statement will transfer control to line number 575, 490, or 650 depending upon whether the value 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 1 or 2.

3.5.6 PAUSE Statement

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

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

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

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
    PAUSED AT 00020 CONTINUE ( Y OR N)? ▷ YES
    THIS IS ANOTHER LINE
    PAUSED AT 00040 CONTINUE (Y OR N)? ▷ NO
*
```

3.5.7 STOP Statement

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

STOPPED AT *line-number*

Programming Note

1. A STOP statement may appear anywhere in the program.

Example:

```
*10 INPUT A
*20 IF A = 1.0 THEN 40
*30 STOP
*40 PRINT "KEEP GOING"
*50 END
*   RUN

?12
    STOPPED AT 00030
*
```

3.5.8 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.

```
*30 END
```

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, then a given sequence of calls to RND will generate the same sequence of numbers for repeated executions.

Example:

```
10 RANDOMIZE
```

3.5.10 TIME Statement

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

```
TIME UP — PROGRAM LOOPING
```

Format

TIME *integer*

The operand of the TIME statement 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 BEM system command.

Example:

```
50 SYSTEM A$
193 SYSTEM "ALLOCATE DA,NEWFILE,PACK02"
```

Programming Notes

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

2. Issuing any of the following commands will terminate the BASIC program: EXEC, LOGOFF, INTR.

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

3.6 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 are required. Included in the group of input/output statements are the INPUT, LINPUT, PRINT, USING, READ, DATA, and RESTORE statements

This section presents these statements in their simple form for use with terminal input/output and program supplied data. A more general (and complete) form of these and several additional statements is presented in Section 4, which describes the use of files.

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 means of 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; the complete input line must be reentered.

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

Example

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

The execution of the above statements would cause the system to type out

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

and 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 statement 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).

3.6.3 PRINT Statement

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

Format

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

where

item — 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 which 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:

—.mantissa 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 I = 1 TO 15
20  PRINT I
30  NEXT I
40  END
```

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

```
Col 1

 1
 Δ1
 Δ2
 Δ3
 Δ4
 Δ5
 Δ6
 Δ7
 Δ8
 Δ9
 Δ10
 Δ11
 Δ12
 Δ13
 Δ14
 Δ15
```

Example 2:

```
10  FOR I = 1 TO 15
20  PRINT I,
30  NEXT I
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         1         1         1         1
 Δ1        Δ2        Δ3        Δ4        Δ5
 Δ6        Δ7        Δ8        Δ9        Δ10
 Δ11       Δ12       Δ13       Δ14       Δ15
```

Example 3:

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

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

```
Δ1ΔΔ2ΔΔ3ΔΔ4ΔΔ5ΔΔ6ΔΔ7ΔΔ8ΔΔ9ΔΔ10ΔΔ11ΔΔ12ΔΔ13ΔΔ14ΔΔ15
```

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

```
20 PRINT — I;

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

Example 4:

```
20  LET A =1
30  C$ = "SALESMAN"
40  A$ = "JOE"
50  B$ = "ΔDOKES"
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  JOEΔDOKES  SALESMANΔ4
```

Example 5:

```
10  PRINT "000000000111111111122222222233333333"
20  PRINT "12345678901234567890123456789012345678"
30  A$="*"
40  A = 1
50  PRINT TAB (10);A
60  PRINT TAB(20);A
70  PRINT TAB(30);A
80  PRINT TAB(10); A$; TAB(20); A$; TAB(30); A$
90  RUN
```

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
!         !           !           !
00000000011111111112222222223333333333
12345678901234567890123456789012345678 9
          1
                      1
                                  1
          .           .           .
```

Example 6:

```
10  FOR I = 1 TO 25
20  PRINT 2**I;
30  NEXT  I
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  3.35544E+07
```

3.6.4 MARGIN Statement

The MARGIN statement is used to set 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 MARGIN statement specifies the largest possible record which can be written to the file.

4. A margin value less than zero, or greater than 4095 is treated as an error.

Example:

```
1     MARGIN     64
```

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

3.6.5 READ and DATA Statements

The READ statement is used to assign to the listed variables values obtained from a DATA statement.

Format

READ *variable* [, *variable* . . .]
DATA *datum* [, *datum* . . .]

where

datum { decimal number | string 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 two blocks of data. Each time a READ statement is encountered anywhere in the program, the appropriate data block supplies the next available datum (or data). The string data block used to supply values for string variables, and the numeric data block is used 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       △.006734   -174.321
3.14159
1
2
4
5
.1234E+17
```

3.6.6 RESTORE and RESET Statements

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

Format

```
RESTORE
RESET
```

Example:

```
 10 READ N
 20 FOR I=1 TO N
 30 READ X
    .
    .
    .
100 NEXT I
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.

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 which 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 above, 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 which 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 his BASIC program. He can left-justify or right-justify the output in the format field defined in the using string.

To left-justify the output, the format field must start with a <. When a format field starts with this character, the field will be 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 will be space-filled. If there are more characters in the string than there are place holders, the string will be truncated.

If the format field starts with a >, then the string will be right-justified in the format field. The last place holder in the field will be replaced with the last character of the string being printed. The next to the last place holder will be 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 > will be replaced by spaces. If the string is longer than the format field, the leftmost characters of the string will be 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 outputted. 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 will cause the dollar sign to be right-justified against the outputted number. The plus sign will right-justify a plus sign against the number if the number is positive, or a minus sign if the number is negative. A minus sign will cause a minus sign to be right-justified if the number is negative; if the number is positive, no sign will be 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 will be 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 would be ****.

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 given above, if the format field used was +##########, the output would be +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 will be known. The +1000000 will be formatted as +1000 E+03 and the +100000000 will be printed as +1000 E+05 which tells the user exactly what was printed. As can be 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 will be adjusted to the field size, and the exponent will hold the magnitude of the adjusted number. If this statement is executed:

```
157 PRINT USING " +##↑↑↑↑↑", 25, 290, −300, .00001
```

the results will be:

```
+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 will begin 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.

There are a few other rules that must also be remembered when printing decimal fields:

- Replace any unused place holders to the right of the decimal point with zeros.
- The maximum number of decimal places that can be printed without an exponent field is six.
- If there are any place holders to the left of the decimal point and the value is less than 1.0, a single zero is printed to the left of the decimal. The sign is justified to this character.

Example:

Number	Format	Result
1.455	+##.##	+1.46
0.50	$##.##	$0.50
0.1234567	+.########	+.123457??

When an exponent file is included with a decimal field, the number is rounded to the proper number of significant digits. It is moved into the format field, and then the proper exponent is calculated and formatted into the exponent field.

Example:

```
10 LET A $ ="+#.#####!!!!!"
11 PRINT USING A$, 1.04
12 PRINT USING A$, 12.345
```

The results will be +1.04000 E+00 and +1.23450 E+01

Another option available with format fields is the choice of + and − signs. These two signs are not equivalent and will produce different output. The plus sign will always cause a sign to be printed in the output field. If the number that is printed is positive, then the printed sign is the plus sign. But, if the printed number is negative, the sign position is replaced with a minus sign. When a minus sign is chosen, it is printed for a negative number, but no sign is printed for a positive number. Also, if the format field that has been defined contains one less place holder than is necessary to print a positive number, then the minus sign will be replaced by the one remaining digit. Thus, the number 100 can be printed in the field −## as 100, but −100 will produce *** as output since the minus sign must also be printed. This allows one-digit positive numbers to be printed in the format field −, and can inhibit the printing of signs in the format field.

The following group of examples is intended to show the user how to use a using string, the errors that can occur, and some practical uses for the PRINT USING format of the PRINT instruction.

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##	2677	TODAY IS THE 26TH OF SEPT, 1977

3.6.7.3 USE WITH THE 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 which 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.

```
242 PRINT A, B; C; D$; USING A$, B, C

243 PRINT #I: USING A$, B, C; D, E$

244 PRINT USING A$, B, C; D, E$

245 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 which 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
```

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

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

Since 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 are formatted. Statement 181 is a normal PRINT statement.

3.7 MATRIX OPERATION STATEMENTS

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

MAT C = A + B	Add the two matrices A and B store the result in matrix C.
MAT C = A — B	Subtract the matrix B from the matrix A and store the result in matrix C.
MAT C = A * B	Multiply the matrix A by the matrix B; store the result in matrix C.
MAT variable = V * W	Multiply vectors V and W and assign the result to a variable.
MAT C = INV (A)	Invert the matrix A and store the resulting matrix in C.
MAT C = TRN(A)	Transpose the matrix A and store the resulting matrix in C.
MAT C = CON	Set each element of matrix C to a value of one.
MAT C = ZER	Set each element of matrix C to zero.
MAT C = IDN	Set the diagonal elements of matrix C to 1's, and all other elements to zero, yielding an identity matrix.

MAT C$ = NUL$	Set each element in matrix C$ to a null string.
MAT C = (exp)*A	Multiply each element of the matrix A by the value of the expression and place the result in matrix C.
MAT INPUT A,A$	Input elements of a matrix.
MAT LINPUT A$,B$	Input lines of data into elements of matrices using the LINPUT statement.
MAT PRINT A,A$	Print elements of matrix A.
MAT READ A,A$	Read elements of matrix A from Data statements.

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 zero are unchanged.

Example:

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

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 which 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., you can't 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 GX2, since the original matrix contained 20 elements and the new matrix would require 28 elements (remember row and column zero).

3.7.2 MAT Addition, Subtraction, and Multiplication Statements

These statements permit addition, subtraction, and multiplication of matrices.

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 operation (*) 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 matrices.

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, if necessary, so that A will be a vector. If both B and C are vectors an error will result. (See 3.7.3.)

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) = current row bound (B)
      current row bound (A) = current row bound (C)

   b. current column bound (A) = current column bound (B)
      current column bound (A) = 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 above 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 (3.7.12) statements 200 through 250 could be replaced by

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

3.7.3 MAT Vector Multiplication

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.

Example:

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

3.7.4 MAT Inversion Statement

Matrices 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 matrix just inverted.
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 bound (B)
   b. current row bound (A) = current row bound (B)
   c. current column bound (A) = current column bound (B)

3. The matrix being inverted will be 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.

3.7.5 MAT Transpose Statement

Matrices 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 will be the transpose of matrix H.

3.7.6 MAT Constant Statement

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

Format

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

where

trimmer is a new array dimension which is 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 one. The dimensions of matrix C are used in the operation.

3.7.7 MAT Zeros (0's) Statement

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

Format

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

where

trimmer	is a new array dimension which is 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:

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

The elements of matrix C will be set to zero. The dimension of matrix C is changed to 3 then the operation is performed.

3.7.8 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 which is 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.

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 will result.

3.7.9 MAT Scalar Multiply

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

Format

MAT *letter = (exp)*letter*

Example:

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

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

3.7.10 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 will be prompted by means of a question mark to enter a list of values. If the array name is 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. If the last array in the list is specified without a trimmer, a variable number of user-supplied values are permitted. The number of values inputted is stored in the function NUM.

Format

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

Example:

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

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 which is not terminated by an ampersand will terminate the input:

   ```
   Line 1: 1, 2, 3, &
   Line 2: 4, 5
   ```

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

3. When doing vector input, the vector is redimensioned to the number of values inputted, in addition to the value being stored in NUM.

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

3.7.11 MAT LINPUT Statement

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

Format

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

Example:

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

3.7.12 MAT PRINT Statement

The MAT PRINT statement causes an entire array (except for row and column zero) 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.

3.7.13 MAT READ Statement

The MAT READ statement causes elements of the matrices 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. Matrices are filled row-by-row until the entire matrix (except for row and column zero) 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 which is to be applied to the matrix.

3.8 PROGRAM SEGMENTATION

The statements described in this section allow BASIC programs to be logically and physically segmented. The CHAIN statement allows a large program to be divided into several smaller ones which may be serially executed occupying the same memory 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 which have been stored in OS/3 library files.

3.8.1 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* / *string-expression* } [WITH *#I* [,*#J*, . . .]]

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 will be assigned sequential channel numbers, beginning at 1. That is, in the chained program, the first file in the list will be assigned to channel 1, the second to channel 2, etc.

Example:

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

Programming Notes

1. If the chained program is specified by a channel expression the file must be a temporary or library file; a data management 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 will be 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 will still be intact.

3.8.2 LIBRARY Statement

This statement is used to inform BASIC of the names of OS/3 library files which 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 will be 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"
```

3.8.3 CALL Statement

The CALL statement is used to invoke a BASIC subroutine.

Format

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

where

string-constant is a subroutine name consisting of at most eight alphanumeric characters.

param-list { *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$, X$(I,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 which are passed on CALL statements must be defined before the CALL statement.

Example:

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

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:

$$\left\{\begin{array}{l} \textit{variable} \\ \textit{channel setter} \\ \textit{function name} \\ \textit{array} \end{array}\right\} \quad , \ . \ . \ .$$

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 will alter the value of that variable; when the caller passes an expression, the parameter will be a local value. The subroutine is not aware of the different parameter modes. However, a returned value would be lost if the subroutine is called with an expression.

2. Channel setter — Any channel constant (#1, #30, etc.). References to this channel will 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 which 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 Note

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(,)
```

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 is used to terminate a subroutine and to return control to the caller. Unlike the SUBEND statement, the SUBEXIT may occur anywhere within the subroutine, except that it may not occur within a user-defined function.

Format

```
SUBEXIT
```

Example:

```
983 SUBEXIT
```

3.9 CHANGE Statement

The CHANGE statement is used to convert arithmetic and alphanumeric formats. It can be used to change a character string into an array of numeric values and vice versa.

Format

CHANGE *string-expr* TO *array* [BIT *expr*]
CHANGE *array* TO *string* [BIT *expr*]

where

string expr	is any string expression which is to be changed.
array	is any numeric array.
expr	is a numeric expression specifying the number of bits per character.

Programming Note

1. In changing from a string to a numeric vector, the BIT expression specifies the number of bits, n, which will be 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 which would not complete a full character, they will be 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 will result.

3. In changing from a vector to a string, the user must set element zero of the vector to the number of vector elements to be converted. Each element in the vector from the first to the last one selected by the user will be converted to a bit string of length n. These bit strings will then form the new string. If element zero contains a zero, 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 will result. An error can also be caused by attempting to create a string greater than 4095 characters.

5. If omitted, the BIT parameter defaults to eight. The maximum permissable 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 file capability in BASIC gives the user a method of saving and retrieving program information permanently. Data in files may be referenced by a program, updated, or new data may be written to the end of a file. The type and format of these files are flexible to enable the user to access files from BASIC and from batch programs. The following sections present the function and format of the statements used to access files, and any special considerations for the use of each file type.

4.2 FILE DESCRIPTION

Three file types are supported by BASIC: Temporary files, Library files, and Data Management files. Although the file types may vary, the actual format of a data record processed by a given statement will not change. This will allow a correctly written program to use the same statement to process a Temporary, Library, or Data Management 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 workspaces. 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 BEM subsystems.

  Since library elements are sequential by nature and may not be extended or updated in place, they are copied to the BEM 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.

- Data Management files

  BASIC permits access to two types of Data Management files: Sequential and Direct Access. Unlike library files, Data Management files do not make use of the workspace, but process the data in place on the disk. Indexed sequential, indexed random, tape, variable blocked, and keyed files are not accessible under BASIC.

  Under most conditions BASIC adapts itself to the file specifications such as record size, block size, and record format, but certain files do not logically permit some operations. The fact that a file is designated sequential or direct access in the Volume Table of Contents (VTOC) does not, itself, determine which BASIC statements will be permitted. This is determined by the block format. For example, fixed length records permit BASIC to locate any record at random given its record number using a simple internal computation, and does not necessitate a search through the entire file. On the other hand, variable length, blocked records do not permit such a computation since the number of records per block varies from one block to the next. Thus a sequential search would be required.

  Any number of data management files may be open simultaneously; however, no more than 29 library and workspace files may be open at the same time.

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 data management), 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 will limit the maximum record size which 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 zero; 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 one, 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 open 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 since 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 zero, which is always defined to be the terminal. Statements such as PRINT, INPUT, and LINPUT may explicitly reference channel zero to access the terminal, but normally no channel setter is specified since the statements default to the terminal.

4.3 FILE STATEMENTS

There are ten BASIC statements used for files. A brief description of each file 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. The program, however, must be carefully written so that it is not restricted to a single file type (e.g., if the program writes 200 character records then it may not be used with Library files since the margin limit of these files is 128 characters).

Table 4-1 BASIC File Statements

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.
MARGIN	All files in BASIC have a margin size which corresponds to the size of the largest record which may be written to that file. The default margin size for all files is 128 characters. The margin size will be set to the record size when a Data Management 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.
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.

(continued)

Table 4-1 BASIC File Statements *(contd)*

File Statement	Use
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.
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.
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.
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 which 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.
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.

(continued)

Table 4-1 BASIC File Statements *(contd)*

File Statement	Use
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. Data management 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.
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.

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. 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 which is being opened. Its exact format varies with the different types of files available.

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 work space. 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 work space when it is opened. Once in the work space, 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. The format of the file name for Library files is:

   element,file[*(readpass/writepass)*][*,volume*]△

   The file parameter (full file definition) must be terminated by at least one space.

4. If the file name specifies an OS/3 Data Management 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. The format of the file name for Data Management files is:

   $\left\{\begin{matrix}SQ\\DA\end{matrix}\right\}$ *,file* [*(readpass/writepass)*] [*,volume*]△

   The file parameter (full file definition) must be terminated by at least one space.

   The user can specify the file parameter by using a variable entry (T$). In this way, the user can type in the file VTOC identifier and choose one of several files on a specified disk pack.

5. Not all Data Management files may be processed. BASIC will process only sequential (SAM) and direct (DAM) access files; it cannot process indexed-sequential (ISAM) files.

6. Data Management files which have variable length, blocked records (VARBLK) may not be read. All other record formats (VARUNB, FIXBLK, FIXUNB) may be used.

7. BASIC will process files with record sizes up to 16K bytes and block sizes up to 65K bytes. Within these limits, any record sizes and block sizes are permitted.

8. Data Management 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 (128), the record and block sizes are set to the margin size, and the file is assumed to have fixed, unblocked records. This is the BASIC default file specification.

9. 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 128 characters, and changed to a library element prior to being closed with the RENAME statement. (See 5.10).

10. 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 couldn't do anything anyway since both READ and WRITE would be inhibited).

Examples:

```
100 FILE #F9:F9$
200 FILE #1:"DATA,BASICLIB,DISK03  "
300 FILE #4000:"*"
400 FILE #D: "SQ,SAMFILE,DISK01  "
500 FILE #10 : N1$ & ",LIBRARY,PACK02  "
600 FILE #47: "DA,PAYROLL(A234/A432)  "
INPUT T$
700 FILE #11:"SQ,"&T$&",DSPOOL "
```

4.3.2 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 (128 characters). Existing Data Management 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 which 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	128 characters
Data Management files	16K characters

4. A temporary file or Library file receives a default margin specification of 128 characters which may be changed at any time after the FILE statement has been issued.

5. The margin size for a Data Management 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.

Examples:

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

4.3.3 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	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 remain 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.

4.3.4 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 would 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 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 may 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 which may be significant. It is important to note that statement 110 was not coded as

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

since 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.

4.3.5 LINPUT Statement

The LINPUT statement allows the user to read in entire records; each record is read into a single string variable. Since 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 #I:  A$
950 LINPUT #I:  B1$  ,  C$(3,4)
960 LINPUT #4:  D$(E+1)
```

4.3.6 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 zero.

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 data management files.

Examples:

```
34 RESET #3: !
35 RESET #4
```

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 which 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 zero, the terminal, may not be referenced by this statement.
variable-name	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 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 zero, 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 data management 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)
```

4.3.9 RENAME Statement

The RENAME statement will change the name of a BASIC file while is it 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 data management 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 wishes 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:

```
1045 RENAME #1:"*"
2074 RENAME #N:"NEW,LIBRARY,PACK09"
```

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 data management 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 data management 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 data management 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 zero.

4. SCRATCH currently has no effect for library files.

Example:

```
104 SCRATCH #6
```

4.3.11 Matrix I/O Statements

To simplify the handling of matrices 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 zero. 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 zero, 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 matrices 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 4.4.

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 4.3.4, 4.3.5, 4.3.7, and 4.3.8, 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 #I: K$,R,A
128 MAT READ #I+1: K$(3),A(3)
129 MAT LINPUT #41: D(2,I)
```

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 BEM monitor. 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 work space, 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 the EDT subsystem.

- DEFINITIONS

  The following syntactic units occur several times in the specification of the editing commands:

1. Line-number	a series of digits in the range of 1 to 99999
2. List-items	line-number line-number — line-number line-number, list-items line-number — line-number, list-items
3. File-parameters	program-name, file-name program-name, file-name (password) program-name, file-name, volume program-name, file-name (password), volume
4. 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 librarian LIBS, OS/3 EDT, OS/3 RSP, 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	OLD
DELETE	PRINT
HELP	RUN
LIST	SAVE
NEW	SYSTEM
modify	

5.2.1 BYE

The BYE command is used to terminate BASIC. Control is returned to the monitor and monitor commands may be entered. All work space information is lost.

Format

BYE

5.2.2 DELETE

This command may be used to delete one or more lines of source from the user's work space. 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. This command should be entered immediately following the message which the user wishes clarified, as the HELP command will always refer back to the last error message.

Format

HELP

5.2.4 LIST,PRINT

The LIST or PRINT command directs the system to display on the user's terminal the lines or sequence of lines referenced in the user's work space. 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

$\left\{\begin{matrix}\text{LIST}\\\text{PRINT}\end{matrix}\right\}$ [*list-items*] [*search-string*]

5.2.5 NEW

The NEW command erases the current contents of the user's work space. The system 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 monitor.

Format

NEW

5.2.6 MODIFY

This command is used to correct or reenter a source statement from the terminal. The statement 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 OLD

The OLD command erases the current contents of the user's work space, then locates and loads the specified program into the user's work space.

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.

5.2.8 RUN

The run command directs the system to compile and execute the program contained in the user's work space.

Format

RUN

5.2.9 SAVE

The SAVE command directs the system to save, on a SAT Library File, a copy of the source program currently contained in its work space. 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:

   OVERWRITE? (Y or 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.

5.2.10 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 work space. Consequently, the source code is not available for editing. Statements are read from the library file, compiled, and written directly into memory. Because this command is intended to be used to execute debugged programs, statement numbers are discarded to conserve memory.

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 work space and an OLD command will have to be issued before the program can be corrected.

3. If execution errors occur, line number zero will be displayed as the error location, because line numbers are not saved during RUNOLD processing.

5.2.11 SYSTEM

This command serves the dual purpose of breaking into BEM system mode without destroying the contents of the work space (compare with BYE) and of providing the ability to execute a BEM 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 [BEM-*command*]

Examples:

```
SYSTEM
SYSTEM STATUS RES
```

5.2.12 MERGE

The MERGE command allows the contents of a library file to be added to the current contents of the work space. Its function is identical to that of the OLD command, except that the work space is not erased first.

Format

MERGE *file-parameters*

Programming Note

1. If lines are read which duplicate the line numbers of lines already in the work space, the new lines replace the old.

5.2.13 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 work space are not modified.

6 BASIC PROGRAM TECHNIQUES

6.1 INTRODUCTION

Constructing a BASIC program requires translation of the problem into a set of statements which 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 and multiline functions, subprograms, and files 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 given as follows:

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: A*B/C** SQR(D) + E

This arithmetic expression is evaluated in the following order:

SQR(D)	Call the result T1 (function)
C**T1	Call the result T2 (exponentiation)
A*B	Call the result T3 (multiplication)
T3/T2	Call the result T4 (division)
T4 + E	Final operation (addition)

In addition, for operators of the same hierarchy the component operations of the expressions are performed from left to right.

Example: A*B/C

This arithmetic expression is evaluated in the following order:

A*B	Call the result T1
T1/C	Final operation

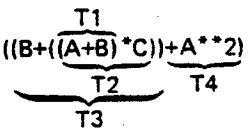
Example: A**B**C

This arithmetic expression is evaluated in the following order:

A**B	Call the result T1
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 + B)	Call the result T1
(T1*C)	Call the result T2
B + T2	Call the result T3
A**2	Call the result T4
(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 simple 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, the above program 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 second example obtains the same table values but with only five 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 to specify a loop even more simply. They are the FOR and NEXT statements and their use is illustrated below:

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

```
10 FOR X = TO 100 STEP 5
```

and in this case the resulting table would contain 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

```
10 FOR X = 100 TO 1 STEP —.1
```

then 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 which 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, then 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 these are referred to as "nested loops." The FOR and NEXT statements may be used for this purpose and these are illustrated in Table 5-1. As can be seen in the table, loops may be nested several levels (maximum of 10), but are never permitted to overlap.

Table 5-1 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 which can be used to designate the elements of a list or a table. These are used where we might ordinarily use a subscript or a double subscript; for example, the coefficients of a polynomial (a_0, a_1, a_2, . . .) or the elements of a matrix b_{ij}. The variables which we use 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 statements:

Example 1:

```
10 FOR I = 0 TO 10
20 READ A(I)
30 NEXT I
40 DATA 2, 3, -5, 7, 2.2, 4, -9, 123, 4, -4, 3
```

Lists and tables whose subscripts exceed 10 require that the user supply a DIM statement to indicate to the system that extra memory space is needed. For example, a list of 15 numbers may be entered as:

Example 2:

```
10 DIM A (25)
20 READ N
30 FOR I=1 TO N
40 READ A(I)
50 NEXT I
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 I = 1 TO 15. However, this program as typed allows for the lengthening of the list simply by changing statement 60, so 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 3 rows and 5 columns could be entered into a program by writing:

Example 3:

```
10 FOR I=1 TO 3
20 FOR J=1 TO 5
30 READ B(I,J)
40 NEXT J
50 NEXT I
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 it 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 will be generated. This is easily rectified by entering the line:

```
5 DIM B(20, 30)
```

if, for instance, a 20 by 30 table is required. 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, DET, LEN, MOD, POS, TIM, VAL, EBC), string functions (CHR$, CLK$, DAT$, SEG$, STR$, USR$), and file-related functions (LOC, LOF, MAR, PER, TYP, 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, 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,ATN(Y4)
70 END
```

In this example, X is $\pi/2$ (90°), Y1 is the sine of 90°, Y2 is the cosine of 45°, Y3 is the tangent of 30°, and Y4 is the arctangent of 15°.

- EXP(X) designates exponentiation e^x.

  Example: 10 D = EXP(X**2)
  In this example, D is e^{x^2}.

- LOG(X) designates the natural logarithm of X, ln(X).

  Example: 10 A = LOG(Y**10)
  In this example, A is 10 ln(Y).

- ABS(X) designates the absolute value of X, $|X|$.

  Example: 10 B = ABS(—X*Y)
  In this example, B is $|-X*Y|$

- SQR(X) designates the square root of X, $\sqrt{X}$.

  Example: 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:

  a. If $X > 0$, then $INT(X) \leq X$
  b. If $X = 0$, then $INT(X) = 0$
  c. If $X < 0$, then $INT(X) \leq X$

Example:

```
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) designates a pseudo random 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 on the open interval [0, 1).

  c. If X = 0, the system supplies a pseudo random 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).

  d. If X is not specified (i.e., RND) then RND(0) is assumed.

To generate a sequence of pseudo random numbers, the user would call any of these options followed by repeated calls to option c.

Example:

```
5  X=0
10 FOR L=1 TO 20
20 PRINT RND(X),
30 NEXT L
40 END
```

- RANDOMIZE may be used to cause RND to supply arbitrary random numbers. It is equivalent to call RND (-1). The execution of the above program would cause the following 20 random numbers to be outputted:

  Example:

```
10      RANDOMIZE
```

```
Col 1          Col 16         Col 31         Col 46      Col 61
↓              ↓              ↓              .           .
.763242E-05    .250198        .753869        .567054     .589602
.747568        .440211E-01    .554667E-01    .252568     .442911
.816485E-01    .52082         .99271         .041932     .572162
.397055E-01    .58698         .801253-       .882914     .793956
```

  If the user wants 20 random one-digit integers, statement 20 could be changed to read:

```
20 PRINT INT(10*RND(X));
```

  This would result in the following output:

```
Col 1                                    Col 78
↓
0 2 7 5 5 7 0 0 2 4 0 5 9 0 5 0 5 8 8 7
```

  The user can vary the type of random numbers desired. For example, if the user wants 20 random numbers ranging from 5 to 24 inclusive, statement 20 could be changed to:

```
20 PRINT INT(20*RND(X) + 5);
```

  In general, if random numbers are to be chosen within the range $A \leq RND(X) < A+B$, then the random function could be used as follows:

```
INT (B*RND(X) + A).
```

- SGN(X) designates the sign of X.

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

  Example:

```
10 X=SGN(0)
20 X1=SGN(-1.82)
30 X2=SGN(X1)
40 X3=SGN(-X1)
```

The execution of this 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.

  Example:

```
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.

  Example:

```
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

  MOD (X,Y) = X - Y (INT (X/Y))

  Example:

```
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$.

  Example:

```
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.

  Example:

```
10    LET A=TIM
20    FOR     I=1 to 1000
30    NEXT I
40    PRINT "ELAPSED TIME IS", TIM-A
50    END
```

  Would print

```
ELAPSED TIME IS        6.325
```

- VAL(Q$) returns the value of the number whose decimal representation is in Q$.

  Example:

```
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) may be used to obtain the EBCDIC value for a single EBCDIC symbol. Certain symbols cannot be typed, and must be entered as 2- or 3-character mnemonics. Lower case lettering may be entered by using the prefix "LC", then the letter to be interpreted as lower case: LCE will be interpreted as an 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.

  Example:

```
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 MM/DD/YY.

  Example:

```
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 06/24/77

- 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$).

  Examples:

  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.

  Example:

```
10    LET N2 = 6.35
20    PRINT  STR$ (N2),  SEG$(STR$(N2), 1,1)
```

  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 a 4-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.

  Example:

```
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 since 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.

  Example:

```
170 FILE #2: "SQ,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: "SQ,ERRORS,SYSRES"
180 A=1
190 IF END #2 THEN 230
200 WRITE #2: A(I)
210 A=A+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 and a +1 returned if the function will be permitted, 0 if not, and −1 if an invalid function statement is used.

  Example:

```
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. In order to avoid program termination should BASIC not permit this, the PER function is used to test if INPUT will be 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$.

  Example:

```
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 which 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.

  Example:

```
* 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 since 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, I
120 REM THIS MULTILINE FUNCTION COMPUTES
130 REM THE FACTORIAL OF N
140 T=1
150 IF N<=1 GO TO 190
160 FOR I=2 TO N
170 T=T*I
180 NEXT I
190 FNA=T
200 FNEND
```

If the above multiline function is called within the sequence of statements:

```
510 FOR J = 0 TO 9
520 PRINT J; "!Δ="; FNA(J)
530 NEXT J
540 END
```

The printed output would appear as follows:

```
Col 1
!
0!Δ=Δ1
1!Δ=Δ1
2!Δ=Δ2
3!Δ=Δ6
4!Δ=Δ24
5!Δ=Δ120
6!Δ=Δ720
7!Δ=Δ5040
8!Δ=Δ40320
9!Δ=Δ362880
```

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 which 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 upper/lower case 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 which use files are presented in this section.

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 in-line.

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 when closed it will be written in place of the original.

Example 3:

```
3000 CALL "FINDNOSP": R1$, C+4, C2
3100 LET C3 = POS (R1$&" "," ",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 to begin the search at, 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 two; the result is returned in parameter three. 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 will improve the execution time and reduce memory 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 matrices 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, combining several values onto one statement rather than one per statement will result in faster RUN compilation.

7 ERRORS AND DEBUGGING

7.1 INTRODUCTION

There are two basic categories of errors: (1) those which prevent the running of the program and (2) those which permit the program to run but cause wrong answers or no answers at all to be printed (these latter errors are referred to as logical errors).

7.2 ERRORS PREVENTING RUNNING OF PROGRAM

It may occasionally happen that the first run of a new program will be free of errors and give the correct answers. But it is much more common that errors will be present and will have to be corrected. The errors in category (1) are detected by the Syntax Checker, the Editing Command Processor, the System Monitor Processor, the System Monitor, the Run-time Error routines, and the Post Compilation routines. (The errors reported by all of the BASIC system components previously mentioned, except the Syntax Checker, are listed in Appendix C. This appendix also contains, 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 on the terminal a question mark (?) followed by a copy of the incorrect statement, up to but not including the first character in error. For example, consider the following statement:

```
10 FOR N=1,
```

Since the comma is not permitted in a FOR statement, the BASIC system responds with

```
? 10 FOR N=1
```

and waits for the user to complete the statement. If the user types in

```
TO 7
```

then the complete statement

```
10 FOR N=1 TO 7
```

is successfully processed by the BASIC system.

7.3 LOGIC ERRORS

Logic errors are those which permit the program to run but cause wrong or no answers at all 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 typing the line number and pressing the TRANSMIT key or by 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, since BASIC arranges them in ascending order once they are read.

As an example, consider the following program which 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 I=2 TO N
40   INPUT X
50   A=A+X
60   IF X>=L THEN 90
70   L=X
80   GO TO 110
90   IF X>=S THEN 110
100  S=X
110  NEXT I
120  A=A/N
130  PRINT "SMALL="; S, "LARGE="; L, "AVERAGE="; A
140  END
```

Assume that when the above program is executed, the user types in the following data values:

```
5,1
2
3
4
5
```

The resulting output would appear as

SMALL=Δ1 LARGE=Δ1 AVERAGE=Δ3

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 to

```
60 IF X <=L THEN 90
```

Once this correction is made and the program is reexecuted with the same input data, the resulting output would appear as

```
SMALL=△1        LARGE=△5        AVERAGE=△3
```

8 BEM OPERATION

8.1 INTRODUCTION

This section instructs the novice user how to use BEM and how to execute BASIC in particular. The purpose of this section is to explain commands available to the terminal user; how to initiate a session, execute and monitor application programs, and terminate the session. For details on how to configure BEM, console operation, etc., the user should consult the *BEM — OS/3 Basic Editor Monitor User Reference*, UA-0139.

To use BEM, the user must locate a free terminal and log on. Logging on consists of entering the LOGON command, together with a user-id, account-number, and password. This user-id is used for identification by the console operator, the account number is used for billing purposes, and the password for security; each is one to four characters long, and either of the last two may be omitted depending on conditions at your site. When the LOGON command is accepted, BEM will display the log-on bulletin and inform the user it is ready to process requests. The log-on bulletin is built by the administrator and may contain messages to inform the users of resource availability or system status.

The administrator may also assign a default file to each user's account, and place certain restrictions on command and file usage. If the user wishes to use the default file, the file and volume names should be omitted from the command which references the file. If a user is informed that he cannot access a certain file, or is not permitted to write to a certain file, the system administrator will need to be contacted to remove the restrictions.

Once logged on, the user is placed in monitor mode and may enter any monitor command. To identify monitor mode, BEM presents the UNISCOPE start of entry (SOE) character, followed by a slash (/). All monitor commands begin with the slash, but if the user does not erase the screen, the slash is supplied by BEM.

Several commands may be entered while in command mode. The HELP command functions identically to the BASIC HELP command. TYPE allows the user to send a message or question to the computer console. Three STATUS commands are provided, one to list users of BEM, another to list OS/3 resources available and those in use, and the last gives information about the user's own terminal.

The EXECUTE command is used to load and run BEM application programs such as BASIC and EDT. The program is located and loaded, and any additional storage (memory or disk) requests for that program are processed. Once loaded, the program is in control of the user's terminal and all key-ins and responses are controlled by it.

While a program is processing, BEM provides a method to interrupt it and return control to the monitor. To interrupt a program, the user hits the MESSAGE-WAITING key or transmits anything. A program may only be interrupted when it is active. If it is awaiting input, the program provides its own way to exit to the monitor; for example, the SYSTEM command may be used to interrupt BASIC.

One advantage of this capability is that during execution a program may be interrupted to enter a monitor command, then the program may be resumed using the RESUME command. The RESUME command will not function if the EXECUTE or LOGOFF commands are entered since the user's work areas are destroyed.

As an example of the interrupt capability in BASIC, the user attempts to RUN a program but is informed that there is insufficient room to load the program. The interrupt capability may be used here to allow the user to enter STATUS commands and wait until sufficient memory is available to load the BASIC program. When it is available, the user may RESUME BASIC, and rerun the program.

At the end of a session, the LOGOFF command is used to release all storage that has been acquired during the session. To begin a session after LOGOFF has been processed, the next user must LOGON again.

8.2 COMMAND FORMAT

All BEM commands begin with a slash (/), and are immediately followed by the command keyword. The slash is normally provided by BEM, but it must be entered if the user has altered the screen, or is operating a non-video terminal. Commands may be abbreviated by typing at least those characters which are underlined.

8.2.1 LOGON Command

The LOGON command is used to begin a BEM session. The "id" used may be one to four characters, and is determined at the user site, and has no actual meaning to BEM.

Format

/LOGON *user-id,[account-number],[password]*

8.2.2 HELP Command

The HELP command allows the user to obtain additional information or explanation about an error or status message which has just been displayed. The HELP command should be entered immediately after the message requiring explanation, since the command always relates to the message immediately preceding the HELP query.

Format

/HELP

8.2.3 TYPE Command

The TYPE command is used to send a message or question to the console operator. All characters following the command keyword are sent to the computer console. Up to 52 characters may be sent.

Format

/TYPE *comment*

8.2.4 PAUSE Command

The PAUSE command is used to send a message or question to the console operator. It is different from the TYPE command in that the user's task is suspended until the operator replies to the message.

Format

/PAUSE *comment or questions*

8.2.5 STATUS Commands

Three formats of this command are available. The first will display information concerning terminals on the system. The second format will display information about OS/3 resources in use by BEM. Lastly, information about the user's own terminal may be obtained.

Format 1

/STATUS TERM

Output Format 1:

TERMINAL	COMMAND	PROGRAM	SCRATCH SPACE	ID
nnnn	cccccc	pppppp	sss	uuuu
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

where

nnnn	Terminal name in the form T*l*t0 where *l* is the line number and T is the terminal number.
cccccc	Last system command issued at this terminal.
pppppp	Last program executed at this terminal.

sss	Number of disk scratch space cylinders acquired by this terminal.
uuuu	User's identification code.

Format 2

/STATUS RESOURCE

Output Format 2:

		----------MEMORY----------			-----SCRATCH-----	
TASKS	TERMS	MAX	AVAIL	FREE	MAX	FREE
nnn	ttt	mmmmm	aaaaa	fffff	sss	ddd

where

nnn	Total number of tasks which may be active at one time; the maximum number of terminals which may be logged on the system.
ttt	Number of terminals currently logged on.
mmmmm	Amount of storage, in bytes, allocated to the entire BEM system as obtained from the job card.
aaaaa	Total amount of storage, in bytes, available for allocation to users and program areas.
fffff	Current amount of storage, in bytes, which is free to be allocated.
sss	Total number of disk cylinders available for allocation to users and programs.
ddd	Current number of disk cylinders which is free to be allocated.

The third STATUS option will display the user's id, terminal number, logon time, current date, and wall-clock time.

Format 3

/STATUS

Output Format 3:

TERMINAL	USER	LOGON	DATE	CUR-TIME
E001	PROC	09:07	78/02/17	09:08:20

8.2.6 EXECUTE Command

This command is used to invoke application programs. Sufficient memory and disk space must be available, if required by the program, for loading to complete successfully.

Format

/EXECUTE *program*

Programming Note

1. Programs which may be executed are EDT, RSP, and BASIC.

8.2.7 LOGOFF Command

The LOGOFF command is used to terminate a session. All work areas assigned to the user are released.

Format

/LOGOFF

8.2.8 FILE STATUS Command

To obtain a directory listing of an OS/3 Library file at the terminal, the FSTATUS command may be used. This command will display the name of each source proc, object, or load module, together with the type of each module. An alternate form of this command (LONG), displays the additional information about each module.

Format

/FSTATUS *library* [(*password*)] [,*volume*] [LONG]

where

library	Name of the file which is to be listed.
password	Read password for the file. It must be supplied with the command if the file was cataloged with a password.
volume	Name of the disk pack on which the file resides. If the file has been cataloged with a volume name, the parameter may be omitted.
LONG	The alternate format of the command is to be used to display the comment, creation date, and time for each module.

This command will produce output similar to the example shown:

```
P-SUPEQU      P-EOJ        S-SRCMOD     S-COPYMOD
S-COBOLPRG    P-CLOSE      O-OBJMOD     L-LODMOD00
```

To obtain a directory listing of the default file for an account (if one exists), enter the command without any operands.

The LONG format of the FSTATUS command produces output similar to:

```
P-SUPEQU      SUPERVISOR EQUATES      02/05/78   12:15
P-EOJ         END OF JOB PROC         01/31/77   02:59
S-SRCMOD      COBOL PROGRAM           07/14/77   14:20
S-COPYMOD     COBOL COPY MODULE       07/14/77   14:35
S-COBOLPRG                            07/14/77   15:05
P-CLOSE       CLOSE THE FILE          01/28/77   22:06
O-LOAD        PROGRAM TO SAVE FILE    09/15/78   08:15
L-LOADMOD     PROGRAM TO SAVE FILE    09/15/78   08:17
```

If the LONG format is used with the default file, a single comma must precede LONG:

```
/FSTAT , LONG
```

8.2.9 PRINT and PUNCH Commands

These two commands may be used to produce a printed listing of a module, or a punched card deck. The PRINT command will list a module on the system printer. A heading identifying the user and line numbers are also produced. The PUNCH command will punch the named module on the system punch. Identifier cards are punched preceding and following each deck to give the user's task information.

Format

/PRINT *element,file* [(*password*)], [*volume*] [,*type*]
/PUNCH *element,file* [(*password*)], [*volume*] [,*type*]

where

element	Name of the module to be printed or punched.
file	Name of the OS/3 Library file which contains the element.
password	Read password for the file. It must be included in the command if the file has been cataloged with a password.
volume	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.

type — Element-type of the module. An "S" denotes source, a "P" denotes proc. If this is omitted, source is assumed.

8.2.10 DELETE Command

This command may be used to delete an element from a library file. Any macro proc, source, object, or load element or group header may be deleted.

Format

/DELETE *element,file* [(*password*)], [*volume*], [*type*]

where

element — Name of the module to be deleted.

file — Name of the OS/3 Library file which contains the element.

password — Write password for the file. It must be included in the command if the file has been cataloged with a password.

volume — 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.

type — Element type of the module:

S Source
P Proc
M Macro
O Object
L Load
G Group header

If this is omitted, source is assumed.

NOTE: Type G specifies that only group headers (BOG and EOG markers) be deleted, not the entire group.

8.2.11 RUN Command

This command will schedule a batch OS/3 job. If a job name is specified, the job control is assumed to be stored in the system Job Control file (YJCS). If the name is omitted, the job control is assumed to be in the JCS queue of the system Spool file.

Format

/RUN *program*

Example:

```
/RUN LISTLIB
/RU
```

When a job is scheduled via BEM, the user will not be notified of its actual initiation or termination. The DISPLAY JOBS command may be used at the terminal to monitor the execution of a batch job.

NOTE: The RUN command is an optional feature of BEM and may not be available at your site due to operating procedures.

8.2.12 DISPLAY Command

The DISPLAY command gives information about OS/3 system usage. This command takes two forms:

- Information about batch jobs
- List of DISK volumes currently mounted

Information about batch jobs may be obtained using the JOBS display option.

Format

DISPLAY JOBS

Example:

```
JOB NAME        SIZE     TIME   STEP   EXEC     JOB NO.
BEM             044986   13.2   01     BEM000   0002
ASMTEST         131072   25.8   02     ASM000   0015
FREE MEMORY     004096
```

where

JOB NAME	The name of each batch job currently executing.
SIZE	Amount of memory allocated to that job, including program load area and job prologue in decimal.
TIME	Current elapsed CPU time for all steps of job, in seconds.
STEP	Step number currently executing.

EXEC.	Name of current load module.
JOB NO.	Unique job number assigned by Spooling.

The unused memory entry shows total free memory at the time of the DISPLAY. Memory allocated to the supervisory, symbionts, and ICAM is not explicitly shown by the display.

A list of disk volumes on the OS/3 system may be obtained with the VOLUMES option:

/DISPLAY VOLUMES

Example:

```
*OS3REL     USER01     *BEMPAK
```

This example shows three disk packs mounted. The two of these that are accessible to the BEM user are marked with an asterisk (*).

8.2.13 SCREEN Command

The SCREEN command is used to inform the BEM system of certain UNISCOPE characteristics or options which the user wishes to utilize.

Format

/SCREEN [*dimension*,] [COP / NOCOP] , [ROLL / NOROLL] , [UTS400]

where

dimension	Size of the UNISCOPE screen: height × width, e.g., 16 × 64, 24 × 80. For a hard copy device, the width is ignored, but the height will control the number of lines printed at a time.
COP	Indicates that all messages output by BEM are to be logged on the COP printer.
NOCOP	Messages are no longer to be logged.
ROLL	All messages displayed by BEM will be displayed at the bottom of the UNISCOPE and the screen will be scrolled up.

NOROLL	UNISCOPE screen will no longer be scrolled.
UTS400	Indicates to BEM that this is a UTS400 terminal and sets the UTS control page for correct RSP operation.

The COP option should not be used unless the device is actually present and configured, or control will not be returned to the terminal. If such a problem occurs, the user should clear the terminal, and issue a /SCREEN NOCOP command to restore operation.

The COP option provides the ability to obtain selected hard copy listings at the terminal. It is not intended to produce a hard copy log of all terminal transactions. Consequently, not all BEM commands will produce meaningful COP listings. To get a hard copy of an FSTATUS or DISPLAY, for instance, the user should format the screen and use the UNISCOPE terminal PRINT button.

The ROLL option truncates all output to a single line on the UNISCOPE screen and thus should not be used when longer lines need to be displayed. Two lines are always left at the bottom of the screen, however, for data entry.

8.2.14 VTOC Command

The VTOC command may be used to display the names of the files on a disk volume. The name of each file on the disk will be shown, along with the number of cylinders allocated to the file, the file type, and extent count. If the file is a library file (file type = SAT), additional information is displayed showing the remaining free space in each partition of the file. This command may be issued to any disk allocated to the BEM system.

Format

/VTOC *volume-name*

Example:

/VTOC PACK22

would produce output similar to:

```
FILE NAME          CYL.   EXTENTS  TYPE  DIRECTORY/  DATA/  B-LOAD

SAM FILE           010      01     SAM
RAND FILE          002      01     D.A.
LIB FILE           050      05     SAT       124/   4021/      4
VERYLONGFILENAME
                   010      02     SAT         0/      0/      0
```

8.2.15 Disk Space Management Commands

These commands allow the user to create and erase files dynamically under BEM. As with most other BEM commands, their use may be restricted by the system administrator for certain accounts.

8.2.15.1 ALLOCATE COMMAND

This command will allocate a new disk file on a specified volume. The file may be any OS/3 file type. If it is a SAT file, it may be initialized as an OS/3 library file.

Format

$$\underline{/ALLOCATE}\ \textit{type,}\ \textit{file-parameters}\ \left[,\underline{INIT} = \left\{\begin{matrix}YES\\NO\end{matrix}\right\}\right]\ [,\underline{SIZE} = n]\ [,\underline{INC} = n]$$

where

type	Indicates the type of file to be allocated: ST — SAT (possibly a library file) IR — IRAM IS — ISAM DA — Direct access SQ — Sequential NI — Non-indexed
file-parameters	Valid OS/3 file description of a file which does not exist on the volume. The volume stated in the parameter list specifies where the file will be placed.
INIT	YES — causes the SAT file to be initialized as an OS/3 library file. This is the default value for a SAT file. NO — the file is not initialized. This is the default value for non-SAT files.
SIZE	Initial allocation SIZE in cylinders. Default value is ten cylinders.
INC	SIZE in cylinders of any extents added when the file is extended. Default is one cylinder.

Any DA, SQ, or NI files allocated may be processed by BASIC. Any initialized SAT file may be processed as a library file by any BEM module.

8.2.15.2 SCRATCH COMMAND

This command will scratch any file except system files. If the file is catalogued, its catalog entry will not be removed. The user should be careful when using this command, as once a file has been scratched, its contents are inaccessible.

Format

/SCRATCH *file-parameters*

where

file-parameters	Is a description of the file to be scratched. This may not be a Y file.

8.2.16 ENTER Command

This command enters an OS/3 library file element to be executed in BEM background mode. This function is only available if it is configured by the system administrator. Tasks entered in background are executed by BEM exactly as from interactive terminals except that output is produced on the high speed printer.

Format

/ENTER *element,file-parameters* [,*type*]

where

element	Is the name of the module to be entered.
file-parameters	Is a description of the OS/3 library file which contains the element.
type	Is the element-type for the module — S denotes source, P denotes proc. If this is omitted, source is assumed.

The ENTER facility allows users to submit an OS/3 library element containing commands and data just as they would be entered at the terminal. This element may contain one or more LOGON-LOGOFF sequences, and each task (LOGON-LOGOFF pair) may perform any functions which would be valid at the terminal. The first statement of an entered deck must be a LOGON command, and there should not be any cards between the LOGOFF and LOGON commands when several tasks are stacked in a single ENTER deck.

Decks submitted via the ENTER function are queued in the OS/3 spool file, along with background decks submitted through the card reader. These decks are then processed in a first-come first-served manner concurrently with interactive processing. The number of tasks available to process these decks is defined by the system administrator; more than one background task may be active at a time.

Output from entered tasks is routed to the main site printers and each task's output is identified with the user-id from the LOGON statement. Invalid LOGON statements in a deck cause BEM to begin rejecting cards until a valid LOGON is found, or the end of the deck is reached. Rejected cards are printed on a separate listing.

Each time an input is expected during a background session, BEM attempts to read the next card. This card could be either a command or a line of data. It is processed just as if it had been entered from a terminal. If an error is encountered during the processing of a command, the error message is printed and processing continues with the next card; the session is not aborted. The only condition which will cause a background session to be aborted is the exhaustion of all input. This is usually due to a missing or misinterpreted LOGOFF statement, and results in the task being logged off.

Certain conditions which normally arise at a terminal have been modified for background tasks:

- CONTINUE queries are eliminated for background tasks and all output is displayed in its entirety. Normally, BEM outputs one screen of lines and suspends output until the user answers the CONTINUE query.
- OVERWRITE queries are eliminated for background tasks. If a module to be written already exists, it is deleted and a new one written automatically.
- OUT OF MEMORY conditions for background tasks are considered errors and a NO response is assumed.
- Batch tasks are treated as hard copy terminals, thus RSP is not available.

8.2.17 COMMENT Command

This command permits the user to enter comments in the comment field associated with an OS/3 library element. The element is located, and then the 30-character comment specified in the command is applied.

Format

/COMMENT *element,file-parameters* [,*type*] *comment*

where

element	Is the name of the OS/3 librarian format element to be commented.
file-parameters	Specifies the location of the file containing the element.
type	Specifies the element type. A P denotes proc or macro; an S or blank, source; an O, object; an L, load.

comment — Is a 30-character string to be used as a comment. It must be separated from the file-parameters by exactly one space. Any additional spaces are considered part of the comment.

8.2.18 BULLETIN Command

This special purpose command allows the system administrator to read, display, and change (using the WRITE keyword) the LOGON bulletin. The READ and WRITE options are restricted to privileged users only, while DISPLAY can be used by any user.

Format

/BULLETIN { READ | DISPLAY | WRITE }

where

/BULLETIN READ — Deletes the entire contents of the user's workspace and then reads the current LOGON bulletin into the workspace. This command should be issued while in EDT or RSP as a SYSTEM command to avoid losing the workspace again on entry into EDT or RSP.

NOTE: This option is equivalent to @DROP; all procs are lost.

/BULLETIN DISPLAY — Displays the current LOGON bulletin to the terminal. This option can be invoked by any user.

/BULLETIN WRITE — Overwrites the existing LOGON bulletin with the contents of the user's workspace.

If the entire new bulletin will not fit in the maximum space reserved for LOGON bulletins, only as much as will fit is written and an error will be displayed. The user can find out how much was accepted via the BULLETIN DISPLAY function. It is allowable to write a new bulletin which is larger than the existing one, provided the maximum bulletin space limit is not exceeded. This command should be issued from EDT or RSP via a SYSTEM command.

8.2.19 RECOVER Command

This command allows the terminal user to recover OS/3 librarian elements which were unintentionally deleted. It is only effective for elements which have been deleted recently and have not been entirely removed from the file via a PAC librarian statement. It must be used carefully to ensure that the correct element is "undeleted" (there may be several to choose from).

Format

/RECOVER *element,file-parameter*[,*type*]

where

element — Is the name of the deleted modules to be recovered.

file-parameters — Is the location of the file containing the elements to be recovered.

type — Is the element type which is to be used to rebuild directory entries for the deleted element.

Once invoked, this command will begin by listing each deleted element which could possibly have the same element type specified in the command. For example, if the user attempts to recover a source module named TEST, and the file contains both source and load deleted modules, only the source modules will be shown:

```
/RECOVER     TEST,MYFILE,MYPACK,S

1. TEST    OS/3    TEST PROGRAM    01/30/78    12:48
2. TEST    OS/3    TEST PROGRAM    01/30/78    14:02
3.* TEST   OS/3    TEST PROGRAM    01/30/78    15:25

SELECT NUMBER AND NEW NAME▷
```

Each element with the name and type indicated will be displayed, with a sequence number for identification purposes. The comment field, date, and time of creation will also be shown. If an undeleted element currently exists, it too will be shown and flagged with an asterisk.

After displaying the list, the user will be asked to select an element (by number), and the name under which the recovered element is to be written. The name selected by the user must not be a name which already exists. As long as this rule is followed, the user may rename any of the modules listed including the active one; thus RECOVER may be used to rename modules too.

Continuing with this example, if the user wished to retain the active TEST element, but also recover deleted element 2, the response

```
▷2,TEST2
```

could be entered to recover copy 2 of TEST and rename it to TEST2. BEM will insure that another module of the same name does not already exist and generate appropriate error messages.

If, on the other hand, the user did not want the active copy of TEST (#3), but wished to restore copy 2, he could rename the active copy and restore copy 2 via:

```
▷3,DUMMY
▷2,TEST
```

and later go back and delete element DUMMY from the file.

Each time the user renames a module, BEM will list the elements again with new numbers to avoid confusion. To end the RECOVER command, type STOP.

8.3 BATCH SUBMISSION

An optional feature available at some sites is the capability for entering card decks of BEM sessions for background execution. This feature permits access to the system when a terminal is not available.

To use the batch capability, the user need only keypunch the session from LOGON to LOGOFF, and submit it to BEM via the computer operator. The deck will be queued and executed on a first-come first-served basis.

Output from batch tasks is routed to the main site printers, and each task's output is identified with the user-id from the LOGON statement.

Batch decks are processed in a manner similar to the decks submitted via the ENTER facility. For additional details on how these decks are processed, and how errors are handled, see the description of the ENTER command (8.2.16).

APPENDIX A SUMMARY OF BASIC STATEMENT AND COMMAND FORMATS

Basic Statement and Command Formats

Operation	Operand Format	Type and Use	Examples
BYE		Command- Terminates BASIC and returns to BEM.	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 #I, #4, #J8
CHANGE	{ *string-name* TO *numeric-vector* / *numeric-vector* TO *string-name* } [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	FN*letter*[$] [(*param-list*)] [, *local-list*] [= *expression*]	Declaration- Defines the entry point into a user function.	54 DEF FND (X,Y) = SQR(X↑2+Y↑2) 55 DEF FNS$ (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)

(continued)

Basic Statement and Command Formats *(contd)*

Operation	Operand Format	Type and Use	Examples
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
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

(continued)

Basic Statement and Command Formats *(contd)*

Operation	Operand Format	Type and Use	Examples
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 #I: D(3,4),J
LET	Format 1: *numeric-variable* [= *numeric-variable* . . .] = *numeric-expression* Format 2: *string-variable* [= *string-variable* . . .] = *string-expression* Format 3: FN*letter*[$] = *expression*	Assignment- Assigns values to numeric or string variables, or to a function.	143 LET A$ = SEG$ (A$,3,4) 145 B(3,4) = SIN(Y) 147 FND = B(3,4) * A(4) + I
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 matrices 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)

(continued)

Basic Statement and Command Formats *(contd)*

Operation	Operand Format	Type and Use	Examples
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 matrices.	190 MAT Q = INV (R)
MAT	*letter* = *letter* * *letter*	Matrix Operations- Multiplies two matrices and places the result in a third.	198 MAT U = V * W 199 MAT A = V * B
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 matrices 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*[$] [(*trimmer*)], ...	Matrix Operations- Solicits input from the terminal or a file and assigns values to each element of the matrix.	253 MAT READ A, B$

(continued)

Basic Statement and Command Formats *(contd)*

Operation	Operand Format	Type and Use	Examples
MAT LINPUT	[*channel-setter* :] *letter$* [(*trimmer*)], . . .	Matrix Operations- Solicits input from the terminal or a file and assigns complete lines of data to each element of the string matrix.	255 MAT LINPUT #3: A$, B$ 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: C, 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 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

(continued)

Basic Statement and Command Formats *(contd)*

Operation	Operand Format	Type and 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)
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
REM	*any characters for a comment*	General Statement- Used for an in-line comment.	391 REM THIS PROGRAM COMPUTES THE EIGENVALUES 392 REM FOR AN ARRAY. 393 REM 394 REMARK
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

(continued)

Basic Statement and Command Formats *(contd)*

Operation	Operand Format	Type and Use	Examples
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" 425 SUB "INTEGRAL" A, FNC, J 429 SUB "FILEFIND" #3, G$
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	[BEM *command*]	Command- Returns control to BEM, or executes a single BEM command without leaving BASIC.	SYSTEM SYSTEM FSTATUS PROGRAMLIB,PACK33
SYSTEM	*string*	Control Statement- Issue a BEM command from a running BASIC program.	475 SYSTEM A$ 476 SYSTEM "RUN" &P1$

(continued)

Basic Statement and Command Formats *(contd)*

Operation	Operand Format	Type and Use	Examples
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 SESSIONS

An example of a complete session is provided in this Appendix to aid the new user in learning BASIC. The designation IN: denotes text, which is supplied by the user, and OUT: designates reponses from the system.

```
 1 IN:    /LOGON USR1
 2 OUT:   USER LOGGED ON, SYSTEM READY
 3 OUT:   /
 4 IN:     EXEC BASIC
 5 OUT:   OS/3 BASIC READY (VER 1.1) 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 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:  NO SUCH LINE NUMBER FOR GOTO OR GOSUB
27 OUT:  LOADER AT LINE 0070
28 OUT:  EXECUTION CALCELLED DUE TO ABOVE ERRORS
29 OUT:  *
30 IN:     LIST 70
31 OUT:    70 IF C$ = "YES" THEN 200
32 OUT:  *
33 IN:     70 IF C$ = "YES" THEN 20
34 OUT:  *
35 IN:     RUN
36 OUT:  PROGRAM TO COMPUTE AREA OF A CIRCLE GIVEN RADIUS
37 OUT:  ENTER CIRCLE RADIUS:▷
38 IN:     1
39 OUT:  AREA OF CIRCLE IS 3.14159     CONTINUE?▷
40 IN:     YES
41 OUT:  ENTER CIRCLE RADIUS:▷
42 IN:     2
43 OUT:  AREA OF CIRCLE IS 12.5664     CONTINUE?▷
44 IN:     NO
45 OUT:  *
46 IN:     BYE
47 OUT:  /
48 IN:     LOGOFF
49 OUT:  USER LOGGED OFF, TERMINAL IS NOW FREE
```

Key for sample BASIC session:

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.
26-29	An error is detected by the compiler at line 70. Execution is inhibited and the user's terminal is returned to compilation mode.
30-35	Line 70 is displayed and the reference to line 200 is corrected to use line 20. Execution is again attempted.
36-37	The program begins execution by displaying a heading line and a request for input.
38	Data is supplied for the INPUT statement at line 30.
39-40	The answer is computed and displayed, along with the question, as to whether to continue or not. The user requests continuation.
41-42	The program again requests input and is supplied a value of 2.
43-45	A second answer is computed and displayed. This time the user selects not to continue the program, and so it terminates.
46	To terminate the BASIC compiler, the BYE command is used.
47-49	A LOGOFF command is issued to end the session.

APPENDIX C SYSTEM ERROR MESSAGES

Error messages appropriate for an interactive environment are short and self-explanatory. For additional information about an error message, the user is directed to the HELP command. The messages are categorized on a functional basis and are listed in the following table. Possible causes of each error and suggested procedures to follow in response to the error are also included. The designation to the right of the message identifies the BEM component which identified the error.

A SUBSTATEMENT OCCURRED BEFORE AN END *BASIC*

Subprograms must occur after the main program. This means that they must be placed immediately after the END statement, or after another subprogram's SUBEND statement.

ACCESS TO PROGRAM NOT PERMITTED FOR USER ID *BEM*

The account currently in use does not permit its users to execute the selected function or program. Use another account, or contact the system administrator to change the account's restriction.

ACCESS TO QUEUE TYPE NOT PERMITTED *RSP*

Access to the queue type (PRINT, PUNCH, READER, LOG, JCS, RBPPU, RBPPR) is not permitted for one of two reasons. The OS/3 Supervisor has not been generated to include the appropriate level of support for the queue, or the BEM job control includes parameters to restrict access to that queue. Consult the system administrator to have the access type changed.

ACCESS TO SYSRES FILES NOT PERMITTED *LIBRARY*

The account currently in use does not permit its users to access files on the OS/3 system pack. Use another account or contact the system administrator to change the account's restriction.

ACTIVE SUBROUTINES EXCEED 16 LEVELS *BASIC*

A maximum of 16 levels of subprogram calls may be issued. Investigate for a possible program loop, or a recursive subprogram call.

ALLOCATE FORMAT ERROR *BEM*

The ALLOCATE command has been entered incorrectly. The format of this command is:

$$/\text{ALLOCATE}\ \ type,file,vol,[\ \text{SIZE}=n\ ][\ ,\text{INC}=n\ ]\left[,\text{INIT}=\left\{\begin{matrix}\text{YES}\\ \text{NO}\end{matrix}\right\}\right]$$

ARGUMENT TOO LARGE FOR EXP(X) FUNCTION *BASIC*

A value has been used with the exponential function which will produce a result greater than the largest number that the 90/30 is capable of handling. The maximum permissible value for the EXP argument is approximately 174.6.

ARRAY SUBSCRIPT OUT OF RANGE *BASIC*

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 upper limit in the dimension statement. If no dimension statement has been used, the upper limit is 10.

ATTEMPTED TO RESET FILE BEYOND EOF OR NEGATIVE *BASIC*

The RESET statement may not reposition the file past the end of the file pointer. The record number specified must be positive.

ATTEMPT TO TEST END OR MORE ON RANDOM FILE *BASIC*

The IF-END or IF-MORE formats may only be used against TERMINAL format files. Check the file type referenced by this statement.

BAD FORMAT — TRY AGAIN *RSP*

The user has transmitted something other than the preformatted parameter table. This may also be the result of using RSP with a UNISCOPE without the protected fields hardware option.

BASIC EDITING COMMAND UNRECOGNIZABLE *BASIC*

Either an invalid command has been entered, or a BASIC statement has been entered without a valid line number. Valid commands are:

OLD	NEW	SAVE
RUN	PRINT	HELP
BYE	DELETE	LIST
SYSTEM	RUNOLD	

BASIC FILE NOT OPEN OR NO DATA STATEMENTS *BASIC*

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 statements are present.

BASIC SOURCE LINES OUT OF ORDER *BASIC*

The lines of source in a BASIC program read in by a RUNOLD or CHAIN statement are not in order by line number. This is mandatory. Do an OLD against the program and then SAVE it.

BEM POINTERS DO NOT AGREE WITH WORKSPACE *WORKSPACE*

The workspace access routines have detected a problem with the in-core and disk pointers. This could have been caused by a previous I/O error, or a modification of the disk by an external source. If the error persists, the user may be forced to halt the current program and reexecute it.

BULLETIN LOCKED — RETRY LATER *BEM*

The bulletin cannot be updated because another user is currently accessing it. Wait until the other user finishes and retry the command. The system administrator should discourage the updating of the bulletin by multiple users.

CHAIN ERROR — INVALID NAME OR PASSING BAD FILE *BASIC*

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.

CHANGE ERROR *BASIC*

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, or invalid string result, or invalid value encountered during conversion.

CHANNEL NUMBER INVALID IN FILE STATEMENT *BASIC*

The channel-setter used in the FILE statement results in a channel number which is not in the range 1 to 4095. Channel zero cannot be defined by a FILE statement.

COMMAND CANNOT BE USED AT THIS TIME *BEM*

A PRINT, PUNCH, DELETE, or FSTATUS command was issued while an active program had been interrupted. The active program was accessing a file at the time of interruption. Allow the interrupted command to complete (RESUME) and then retry the command.

COMMAND KEYWORD OMITTED *EDT*

An operand has been entered for which there is no command function. For example, the file parameters have been used without the specification READ or WRITE.

COMMAND TERMINATED *EDT*

The EDT command which was active when the user issued a /INTR or'DISCONTINUE command has been terminated. Informational message only.

CONTINUE? (Y OR N) *ALL*

BEM has displayed a full screen or page and has additional output for the terminal. When ready, the user may respond with a Y to see additional displays, or an N to terminate the display and the command. A response other than Y or N will result in the CONTINUE message being displayed again.

COPY WITH NUMBER OPTION INVALID *EDT*

The COPY command may not be used with the NUMBER command.

DEF MUST PRECEDE REFERENCE IF LOCALS ARE USED *BASIC*

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.

DESEQ OPTION ONLY ALLOWED WITH READ *EDT*

The DESEQUENCE option is only meaningful when used with the EDT READ command; in all other cases its use is treated as an error.

DEVICE UNAVAILABLE AT THIS TIME *BEM*

The printer or punch is not configured and may not be used. Contact the system administrator to have the printer or punch configured.

DIMENSIONS INCONSISTENT IN SUB CALL *BASIC*

The type of variables used in the SUB and CALL lines differ. Either a scalar variable was used where an array was expected, or the number of subscripts on the SUB and CALL lines differ.

DISPLAY COMMAND PARAMETER ERROR *BEM*

The DISPLAY command has been entered incorrectly. Valid options are:

/DISPLAY {JOBS / VOLUMES}

DIVISION BY ZERO, EXECUTION CONTINUES *BASIC*

The program has attempted a division by zero. The algebraic result of division by zero is undefined; however, execution continues using a high value.

EDT VARIABLE AREA NOT AVAILABLE *EDT*

There is currently insufficient storage available to use EDT variables.

ELEMENT/GROUP NOT FOUND *BEM*

The element or group specified in the DELETE command could not be found. Check the spelling of the name and check the names in the file via FSTATUS.

ELEMENT IS NOT IN THE LIBRARY FILE *LIBRARY*

The program requested by the command is not 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).

ELEMENT NUMBER DOES NOT EXIST, RE-ENTER ⊳ *BEM*

The user did not select one of the numbers listed by the RECOVER command. Only those elements identified with a number in the left margin may be recovered. Reenter the correct number and the new module name.

END OF FILE ON INPUT OR LINPUT *BASIC*

The program has issued an INPUT or LINPUT statement which attempted to read more records than were in the file. Investigate the program logic to determine why too many records are being read.

END STATEMENT IS MISSING OR MISPLACED *BASIC*

All BASIC programs must have an END statement as the last line. Insert an END statement and rerun.

ENTER ELEMENT NUMBER, NEWNAME OR "STOP"⊳ *BEM*

The RECOVER command has presented a list of elements which could be recovered. Select one by specifying its number, and a new name for it. Other possible responses at this point are STOP to terminate the command, or HELP to obtain additional information.

ENTER FILE NAME *BASIC*

The user has entered a SAVE, OLD, or RUNOLD statement without specifying a file name. Supply the name in response to this message.

ENTER FUNCTION NOT CONFIGURED *BEM*

The system administrator has not elected to make the ENTER command available at your site. Contact the administrator to have the function installed. This error may also be the result of not having OS/3 Spooling configured, or not having any spooled Input Readers.

ERROR IN READING CARDS/ENTER STREAM, USER CANCELLED *BEM*

A fatal I/O error has occurred while reading cards from a batch stream or enter file. The batch is discarded and the user is cancelled.

ERROR IN READING SCRATCH SPACE *WORKSPACE*

An I/O error has occurred while reading from the work area. Retry input or investigate for possible hardware problem.

ERROR IN READING SCRATCH SPACE INDEX *WORKSPACE*

An I/O error has occurred while reading the work area index. Retry input or investigate for possible hardware problem.

ERROR IN SOURCE — RESEQUENCE TERMINATED *BASIC*

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 resequenced. This error indicates that there is at least one statement which is not syntactically correct.

ERROR ON READ FROM FILE (INVALID NUMBER) *BASIC*

A READ statement attempted to read a numeric variable. The record which was read did not contain numeric data.

ERROR PROCESSING USER FILE LABELS *FILES*

The file being accessed contains user file labels. These cannot be processed by BEM.

ERROR WHILE WRITING INTO SCRATCH SPACE *WORKSPACE*

An I/O error has occurred while writing to the work area. Retry input or investigate for possible hardware problem.

EXPONENT OVERFLOW, EXECUTION CONTINUES *BASIC*

The result (or intermediate result) of a computation has exceeded the largest number the 90/30 is capable of handling. This number is approximately 10^{75}. Machine infinity is supplied and execution continued.

EXPONENT UNDERFLOW, EXECUTION CONTINUES *BASIC*

The result (or intermediate result) of a computation is less than the smallest number the 90/30 is capable of handling. The number is approximately 10^{-78}. Zero is supplied and execution continued.

EXPONENTIATION ERROR *BASIC*

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.

EXPRESSION OUT OF COMPUTED GOTO RANGE *BASIC*

The calculated expression is not a valid number for this computed GOTO. It is either too large or nonpositive. The count of line numbers in the statement determines the largest value the expression may have.

FILE ACCESS HAS BEEN TERMINATED BY USER *LIBRARY*

This indicates that a file access has been terminated when the user did not wish to wait on a FILE IS IN USE message.

FILE ALREADY EXISTS ON VOLUME *BEM*

The user is attempting to allocate a file which already exists on the specified volume.

FILE DOES NOT HAVE VALID "ENDLIB" *LIBRARY*

While searching the directory of the file, BEM could not find the ENDLIB marker. The file's integrity is in question. A possible solution would be to copy all elements to another file, then scratch and rebuild the original file.

FILE IS EMPTY — ENDLIB MISSING *LIBRARY*

The user has attempted to access an empty library file. Initialize the file with the Librarian in order to use it with BEM.

FILE IS IN USE, PLEASE WAIT *LIBRARY/FILES*

Another user is accessing the file. After his command completes, yours will begin. If you don't wish to wait, interrupt the system.

FILE IS NOT AN OS/3 LIBRARY FILE *LIBRARY*

The file specified by the command is not a library file, or has not been initialized by the librarian. Have the system administrator prepare the file, and be sure you are using the correct file.

FILE PARAMETERS DO NOT FOLLOW "FSTATUS" *BEM*

The FSTATUS command requires file parameters in the format:

filename (password), volume

FILE PARAMETER FORMAT ERROR *LIBRARY/FILES*

The file parameters given for a file-access function are not valid. The maximum length for each parameter is: name, 8; filename, 44; password, 6; volume, 6. If a module type has been supplied, it must be S, P, or M.

FILE REQUESTED IS NOT ON DISC VOLUME *LIBRARY/FILES*

The filename requested is not on the volume specified. Check the spelling of the filename or verify that the file is on the volume.

FILE STATEMENT INVALID FOR # 0 *BASIC*

The channel-setter specified with the FILE statement results in a value of zero. Channel zero, the terminal, cannot be defined by a FILE statement.

"FNEND" FOUND WITHOUT FUNCTION DEFINITION *BASIC*

The FNEND statement was detected, but it was not at the end of a function. Remove the statement or place it in the correct location and rerun.

"FNEND" STATEMENT MISSING *BASIC*

A user-defined multiline function exists in the program without a closing FNEND statement. Locate the function and insert the FNEND statement.

FUNCTION ASSIGN DOES NOT MATCH FUNCTION NAME *BASIC*

The name of the function being assigned differs from the name of the function in which it appears. Only the function being defined may be assigned a value.

FUNCTION ASSIGNMENT MUST APPEAR WITHIN FUNCTION *BASIC*

A value must be assigned to a multiline function before the FNEND statement. The function value may not be assigned outside the body of the function.

FUNCTION DEF MUST PRECEDE USE IN "CALL" *BASIC*

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.

FUNCTION DEFINITION WITHIN A FUNCTION *BASIC*

BASIC has detected a function within the body of another function definition. Check for a missing FNEND statement or restructure the function.

FUNCTION EXPECTED IN CALL OR SUB LINE *BASIC*

A previous CALL statement passed a function reference. This CALL did not pass a function. The parameter types must be the same. Resolve the conflict and rerun the program.

FUNCTION HAS NOT BEEN DEFINED *BASIC*

The function referenced on the line in error has not been defined. Define the function or remove the reference to it and rerun.

GIVEN LINE EXCEEDS 80 CHARACTERS WHEN RESEQUENCED *BASIC*

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 in length.

GOTO INTO OR OUT OF FUNCTION DEFINITION *BASIC*

A function may not reference program lines which do not occur within the body of the function, nor may statements outside the function reference lines within the function body. This applies to GOTO, GOSUB, ON, and IF statements.

ICAM ERROR (INPUT TOO LONG) RETRY *BEM*

The last message sent from the terminal to BEM did not arrive correctly; retransmit it. Any input to BEM is limited to 128 characters in length. If this error is displayed while transmitting the RSP Spool file descriptor screen, it indicates the UNISCOPE being used does not have the required protected format feature.

ILLEGAL COMBINATION — "NOT" INVALID *EDT*

If the NOT option is specified, then a search-string must also be specified and a change-string must not be specified.

ILLEGAL COMBINATION OF COMMANDS *EDT*

Several command keywords have been entered which conflict. See Table 3-1 in UA-0141 for allowable combinations.

ILLEGAL "VAL" ARGUMENT *BASIC*

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 prefixed or suffixed to the number.

INCORRECT NESTING OF FOR-NEXT STATEMENTS *BASIC*

A FOR or NEXT statement, which was not nested correctly, was detected. Possible causes are:

1. A FOR statement that has the same index as a previous FOR statement in the nest.
2. A NEXT statement that does not have the same index as the FOR statement immediately preceding it.
3. A NEXT statement that does not follow any open FOR statement.

INCONSISTENT FORMAT IN "USING" STRING *BASIC*

The format field type does not match the type of variable being printed. Either a string was printed into a field beginning with $, + or −, or a number was printed into a field beginning with < or >.

INPUT DATA INCORRECT, RE-ENTER *BASIC*

The data entered for an input statement does not match the variable types required by the program. The entire line must be reentered. This error message could also be caused by too much or too little data in the input response.

INSERT ERROR (DUPLICATE OR INVALID CHANGE STRING) *EDT*

Either the keyword INSERT is preceded by a string, or it is not followed by one. The change string may also be invalid. See Section 3.1.2.4 or 3.1.1.6 of UA-0141.

INSUFFICIENT DATA TO READ *BASIC*

All DATA statements in the program have been used, yet the program attempted to request additional data.

INSUFFICIENT INFORMATION TO CREATE SPOOL FILE *RSP*

The minimum information required to create a Spool file was not specified on the spool descriptor screen. An input file requires either an LBL, or both a JOB NAME and an LFD.

INSUFFICIENT RESOURCES TO LOGON *BEM*

There is not enough memory or user tasks to allow another user to log on to the system. The user should wait until another user has released storage or logged off.

INTERNAL ERROR IN LIBRARY ACCESS ROUTINE *LIBRARY*

The library access routine within BEM has detected a logic error. Take a dump as soon as possible, save all relevant data, and consult your Sperry Univac representative.

INTERNAL ERROR IN RESEQUENCE ROUTINE *BASIC*

A condition which should not normally exist has been detected by the resequence routines. Collect all relevant data, obtain a memory dump, and contact your local Sperry Univac customer representative.

INTERNAL ERROR IN WORKSPACE *WORKSPACE*

An internal error has been detected by the workspace access routines in BEM. If the error persists the user may be forced to halt the current program and reexecute it.

INTERRUPTED: (C)ONT,(D)ISCONT,(S)YSTEM⊳ *BEM*

This message indicates that the user has interrupted BEM by means of the MESSAGE-WAITING key on a UNISCOPE terminal, or the BREAK key on a hardcopy terminal. The user has three options: C- will continue the interrupted operation; D- will discontinue the current operation and return to command mode; S- will temporarily suspend the current operation and allow the user to enter BEM commands; when the user wishes to resume the current operation, the /RESUME command is used.

INVALID @(LABEL) -- MISSING PAREN *EDT*

An open parenthesis was found to start a label, but there is no closing parenthesis.

INVALID @SET COMMAND *EDT*

An @SET command has been used with an invalid keyword parameter. The only valid keywords for use with SET command are PAGE, LINE, TABS, and CHAR. See Section 3.2.3 of UA-0141.

INVALID ASSIGN STATEMENT *EDT*

An ASSIGN statement must be of the form:

@ASSIGN G*n* = *expression*

INVALID BLOCKSIZE OR RECORD SIZE *FILES*

BEM 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 zero, or greater than 65K.

INVALID BULLETIN OPTION — NOT READ/WRITE/DISPLAY *BEM*

There are only three valid bulletin functions which may be used with the BULLETIN command. These are READ, WRITE, and DISPLAY. Correct the command and retry it.

INVALID BY PARAMETER USAGE *EDT*

The BY specification has been used without the SEQUENCE command, or the form of the parameter is not valid. See Section 3.1.1.2 of UA-0141.

INVALID CHANNEL SET EXPRESSION *BASIC*

The channel-setter in the flagged statement resulted in a number less than zero, or greater than 4095. Channel numbers must be between 0 and 4095.

INVALID COLUMN IN TAB COMMAND *EDT*

One of the column numbers used in a TAB command is not between 1 and 128. See Section 3.2.3 of UA-0141.

INVALID COLUMN RANGE *EDT*

The column range specified in a replacement expression is invalid. It must be of the form:

n:i-j

where $1 \leqslant i \leqslant j \leqslant 128$

INVALID DO OPTION *EDT*

The DO statement has the format:

@DO n [P]

where n is an integer (1-9) and the optional P specifies that each command is to be printed.

INVALID EDT VARIABLE (#Gn) *EDT*

A single number sign (#) is assumed to designate an EDT general variable. This must be followed by the letter G and a digit in the range 0-9. If a number sign is needed in the command, enter two number signs (##).

INVALID ELEMENT TYPE *BEM*

The element type used with the /DELETE command must be one of the following:

S-source	P-proc	M-macro
O-object	L-load	G-group

INVALID EXPONENT FIELD IN USING STRING *BASIC*

An exponent field must consist of exactly five up-arrows (↑) and cannot be followed by a place holder #. Correct program and rerun.

INVALID FIELD DESCRIPTOR, EXPECTING <, > *BASIC*

The user program attempted to print a string variable with a numeric format. Correct the program and rerun.

INVALID FIELD DESCRIPTOR, EXPECTING $,+,— *BASIC*

The user program attempted to print a numeric variable with a string format. Correct the program and rerun.

INVALID FORMAT FOR LOGON COMMAND *BEM*

The LOGON command has been entered incorrectly. It must begin with the word LOGON, and is followed by one to three fields of up to four characters each. Check that none of the fields are too long, and that there is nothing entered after the third field.

INVALID ID, ACCOUNT, PASSWORD FOR LOGON *BEM*

An unlisted id, account, and password combination has been entered; thus the user has been denied access to the system. If the fields have been entered correctly, then the account may have been removed from the system. Contact the system administrator to have the account created.

INVALID IF STATEMENT *EDT*

An IF statement has the following format:

@IF *expression op expression COMMAND*
@IF {T. / F.} *COMMAND*

INVALID KEY LENGTH *FILES*

Files containing keys cannot be processed by BASIC.

INVALID LINE RANGE *BASIC*

Valid line ranges consist of single line numbers (a,b) or ranges of lines (a-b). A line number consists of a decimal number in the range 1-99,999.

INVALID LINE SET COMMAND *EDT*

An at sign (@) alone has been entered, or the line number with the line set command is not valid. See Section 3.2.4 of UA-0141.

INVALID OR ZERO LINE NUMBER *EDT*

A line number in an EDT variable expression must be in the form nnnn.nnnn, and must not be zero.

INVALID MAJOR FRAME COMMAND *RSP*

The user has entered a command other than one of those shown on the screen. A new screen will be presented. Valid commands are:

BREAK	END	DISPLAY	RETRIEVE
RELEASE	BUILD	DELETE	HELP
CLEAR	READ	WRITE	SCREEN
TYPE	UPPER	LOWER	SYSTEM

INVALID MARGIN SIZE *BASIC*

The margin expression specified on the flagged statement resulted in a number less than zero, 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.

INVALID NUMBER PARAMETER *EDT*

The number parameter must follow the NUMBER command, must be a valid change string, and must terminate with at least 1 but not more than 15 numeric characters. The parameter must be enclosed in apostrophes and any "or" characters in the string must be entered twice.

INVALID OR DUPLICATE CHANGE STRING *EDT*

The change-string used is not valid or two change-strings have been entered. Change-strings must begin and end with apostrophes. See Section 3.1.1.6 of UA-0141.

INVALID OR DUPLICATE COLUMN RANGE *EDT*

The column range entered is not valid, due to incorrect format, or two column ranges have been entered. See Section 3.1.1.3 of UA-0141.

INVALID OR DUPLICATE COPY-TO LOCATION *EDT*

Either two copy-to locations have been used (i.e., @ COPY 1-10 TO 11 TO 22) or the one given is not valid. The number following the word TO must be a valid line number. See Section 3.1.1.7 of UA-0141.

INVALID OR DUPLICATE LINE RANGE *EDT*

The line range entered is not valid due to incorrect format or two line ranges have been entered. See Section 3.1.1.9 of UA-0141.

INVALID OR DUPLICATE SEARCH STRING *EDT*

The search-string used is not valid or two search-strings have been entered. Strings must begin and end with quotes or apostrophes. See Section 3.1.1.8 of UA-0141.

INVALID PROC GROUP NUMBER *EDT*

The PROC group number must be a single digit integer in the range 1-9.

INVALID RESPONSE, ENTER NUMBER, NEWNAME ▷ *BEM*

The user's response to the last query was incorrect. A non-zero number must be entered first, followed immediately by a comma and then the module name. No intervening spaces are permitted.

INVALID SCREEN ROLL COMMAND *RSP*

The user has entered a command other than one of those shown on the top of the screen. A new screen will be presented. Valid commands are:

CMD	UP	DOWN
RIGHT	LEFT	DELETE
INSERT	UPDATE	REFRESH

INVALID SEARCH COMMAND *RSP*

The search command issued to RSP is not correct. It consists of a search-string and an optional column range. The search-string must begin and end with an apostrophe. A column range is a single number, or two numbers separated by a hyphen. The number must be between 1 and 256.

INVALID SEARCH STRING *BASIC*

A search-string consists of any character string enclosed in quotation marks. If a quote appears in the string, it must be typed as " ".

INVALID SUBSTRING EXPRESSION *EDT*

A substring of an EDT variable is written as a starting position (s) and a length (l) enclosed in parentheses--(s,l).

where $1 \leqslant s \leqslant 50$ and $s + l \leqslant 51$

INVALID TAB EXPRESSION FOR PRINTING *BASIC*

The argument of the TAB function was less than one.

INVALID TRIMMER IN MATRIX STATEMENT *BASIC*

Either the trimmer specified did not result in a positive number, or the resultant array would require more storage than the original array.

INVALID VARIABLE EXPRESSION *EDT*

An EDT variable expression must be one of:

STRING — 'ABC'
VARIABLE — Gn
NUMERIC EXPRESSION — n + m, n − m, n
LINE/COLUMN RANGE — n:i−j

I/O AREA COULD NOT BE LOCATED, RETRY *LIBRARY*

An I/O area for the library function could not be acquired by BEM. Wait a few minutes and retry. If the problem persists, contact the system administrator to have the system memory partition enlarged.

I/O ERROR ACCESSING MESSAGE INDEX *BEM*

An I/O error has occurred while writing to the bulletin file. Only part of the bulletin is now valid. The DISPLAY option should be used to determine that status of the bulletin, and the WRITE option may then be retried.

I/O ERROR ON WRITE TO FILE *FILES*

An I/O error has occurred while writing the data management file. Investigate for possible hardware problem or retry the program.

I/O ERROR WHILE ACCESSING V.T.O.C. *LIBRARY/FILES*

An I/O error has occurred while accessing the VTOC for the disk volume specified. Retry or investigate for possible hardware problem.

I/O ERROR WHILE READING CATALOG *LIBRARY/FILES*

An I/O error has occurred while reading the catalog. Retry or investigate for possible hardware problem.

I/O ERROR WHILE READING LIBRARY FILE *LIBRARY*

An I/O error has occurred while reading the library file. Part of the program may be missing. Retry or investigate for possible hardware problem.

I/O ERROR WHILE WRITING LIBRARY FILE *LIBRARY*

An I/O error has occurred while writing the library file. The program has not been saved. Retry the command or investigate for possible hardware problem.

LIBRARY FILE FULL, ELEMENT NOT ADDED *LIBRARY*

The library file has been filled and there is not enough room to write out the program. The old version, if any, is left intact. Have the file expanded or its contents compressed.

LIMIT OF 4 "LIBRARY" STATEMENTS EXCEEDED *BASIC*

BASIC will search at most four libraries for subprograms, the program has attempted to use more than four.

LOADER AT LINE xxxxx *BASIC*

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.

LOG OF A NON-POSITIVE NUMBER UNDEFINED *BASIC*

The LOG function has encountered a nonpositive argument. The logarithm of a nonpositive number is undefined, thus execution is cancelled.

MATRIX DIMENSIONS ARE INCORRECT FOR FUNCTION *BASIC*

The row or column dimension of the matrices in the matrix statement is incorrect. Check DIM statement for the matrices in question.

MISSING FILE PARAMETER *EDT*

A READ or WRITE command has been entered, but the file parameters do not immediately follow the command keyword. Correct and retry.

MISSING FILE PARAMETER *BEM*

File parameters must immediately follow the DELETE, PRINT, or PUNCH keyword and be in the format:

element, filename (password), volume, type

MODULE NOT OVERWRITTEN, COMMAND TERMINATED *LIBRARY*

This is a confirmation message informing the user that the WRITE command was not executed. It results from a NO answer to the OVERWRITE question.

MORE THAN 29 FILES OPEN *BASIC*

BASIC does not support the concurrent use of more than 29 temporary and library files per user. This program has exceeded the limit.

MORE THAN 4 CHARACTERS IN LABEL *EDT*

EDT statement labels may contain no more than four characters. Correct the proc by using shorter labels.

MUST BE PRIVILEGED FOR BULL READ/WRITE *BEM*

Only privileged users may read or write the BEM bulletin. Normally, only the system administrator will have a privileged status in the accounting file.

NEW NAME ALREADY EXISTS, RE-ENTER▷ *BEM*

An element with the specified name already exists in the file. Select another name and retry the response.

NO DISK SCRATCH SPACE AVAILABLE *WORKSPACE*

All the disk cylinders available to BEM have been assigned, wait and retry or contact the system administrator.

NO FORMAT STRING DEFINED IN USING STRING *BASIC*

The user program attempted to print a variable using a format string that does not contain any valid format strings.

NO MEMORY AVAILABLE FOR FILE I/O BUFFERS *FILE*

A memory area to store a block buffer and DTF could not be allocated for your file. Retry later and contact the system administrator if the problem persists.

NO MEMORY AVAILABLE FOR WORKSPACE BUFFERS *WORKSPACE*

An area of memory could not be acquired for I/O buffers. Retry command when memory becomes available. If problem persists, contact system administrator to have the memory partition size increased.

NO PROC TO END *EDT*

The @END statement was issued while no proc was active.

NO SUCH LINE NUMBER FOR A GOTO OR GOSUB OR IF-THEN *BASIC*

The line number referenced in a GOTO, GOSUB, ON, or IF-THEN statement is not present in the program or function. Insert the required statement or remove the reference to it.

NOT A DATA MANAGEMENT FILE *FILES*

The file being accessed is not a valid sequential or direct access file. To be valid, it must have type SQ or DA and contain a single partition.

NOT ENOUGH MEMORY IS AVAILABLE TO LOAD *BEM*

Insufficient storage is available to load the function you are calling for. Wait and retry. If the problem persists, contact the system administrator to have the memory partition size increased.

NOTHING HAS BEEN FOUND TO RECOVER *BEM*

The RECOVER command has searched the library for deleted modules with the same name and type as specified in your command, but could not find any. This response may indicate that the library has been packed, or that you have recovered all deleted elements and there aren't any left to display.

NULL USING STRING NOT ALLOWED *BASIC*

The using string specified in the print statement is a null string. Define the variable and rerun.

NUMBER OF ARGUMENTS INCONSISTENT *BASIC*

The number and type of arguments passed in the CALL statement(s) do not agree with the number and type on the SUB line.

NUMBER OF PARAMS IN FUNCTION CALL INVALID *BASIC*

A maximum of 16 passed parameters and local variables may be specified on a function declaration line. Reduce the number of labels and rerun.

NUMBER OF SUBSCRIPTS FOR ARRAY INCORRECT *BASIC*

The variable that caused the error has been dimensioned with a different number of subscripts than were found in the reference to it.

OPERATION NOT PERMITTED TO FILE *BASIC*

The operation to be performed against the file conflicts with the file type.

OS/3 ALLOCATE ERROR *BEM*

This message is returned when the ALLOCATE command receives an error status from the supervisor when trying to allocate the file. It may indicate that there is insufficient space on the disk volume.

OUT OF MEMORY - RETRY (Y OR N) *ALL*

One of the internal routines within BEM has attempted to acquire additional storage on a temporary basis. No storage was available. The user may wait for storage to become available and reply Y, or may terminate the current program by replying N. If the problem persists, contact the system administrator to have the memory partition size increased.

OVERFLOW ON VARIABLE SUBSTITUTIONS — TRUNCATED *EDT*

When variables in a command line were replaced, the new line exceeded 80 characters. The truncated command was processed.

OVERWRITE? (YES OR NO) *LIBRARY*

The program to be written out by the command already exists on the file. A reply of YES will overwrite the previous version with the new one. A reply of NO will terminate the command.

PAGE/LINE SIZE INVALID *EDT*

The page or line sizes are not within the correct range. PAGE must be between 1 and 255, LINE must be between 1 and 128.

PARAMETER TYPE MIS-MATCH *BASIC*

The type of a parameter passed to a function/subprogram conflicts with the type defined for the function subprogram. For example, a string was passed when a numeric value was expected or a numeric value was passed when a string was expected. Compare the line in error and the definition; correct the discrepancy.

PASSWORD IS INVALID FOR FILE *LIBRARY/FILE*

The password used does not match the one cataloged for the file. Another cause of this error could be failure to specify a password with the file-access command. The user is denied access to the file in either case.

PAUSED AT xxxxx CONTINUE (Y or N) *BASIC*

A PAUSE statement has been encountered at the line number given by *xxxxx*. Answer YES to continue execution; answer NO to terminate the program.

PLEASE LOGON *BEM*

The user's terminal has not been joined to the BEM system. Follow log-on procedures given in Section 2 of UA-0139.

PRINT TO FILE > MARGIN SIZE *BASIC*

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.

PRINT/PUNCH I/O ERROR *BEM*

A hardware I/O error has been encountered on the printer or punch. Retry the command. If the problem persists, investigate a possible hardware error.

PRINTER/PUNCH IS IN USE, PLEASE WAIT *BEM*

Another user is using the printer or the punch. Your command will be completed after the other command completes. If you do not wish to wait, interrupt the system.

PROGRAM CANNOT BE RESUMED *BEM*

The user tried to resume a program when no program had been loaded. A RESUME command is only effective when the user has interrupted an active program and wishes to return to it.

PROGRAM COULD NOT BE FOUND *BEM*

The program to be executed via an EXECUTE command could not be found. Only EDT, RSP, and BASIC may be loaded under level 4.0.

PROGRAM NOT INCLUDED IN CONFIGURATION *BEM*

The system administrator has not elected to provide the program you have requested.

REFERENCE TO ACTIVE PROC *EDT*

The user has attempted the DO option on a currently active proc, or has attempted to enter the current proc with an @PROC command.

REFERENCED SUBROUTINE NOT FOUND IN LIBRARIES *BASIC*

All user-specified libraries have been searched, but the subprogram listed in the error message could not be found. Execution is inhibited.

RENAME ERROR *BASIC*

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.

REQUESTED RECORD NOT FOUND IN DATA FILE *FILES*

BASIC was attempting to read a record which does not exist in the data management file. Probably due to a hardware error. If the problem persists, consult your Sperry Univac representative.

RETURN WITHOUT MATCHING GOSUB CALL *BASIC*

The program has attempted to return from a subroutine that was not called by a GOSUB statement.

ROLL OPTION VALID ONLY AT UNISCOPE TERMINALS *BEM*

The /SCREEN ROLL or /SCREEN COP options cannot be used with a hardcopy terminal. The command is ignored.

RSP AVAILABLE ONLY AT UNISCOPE TERMINALS *RSP*

RSP cannot be used at a hardcopy terminal. Move to a UNISCOPE terminal and reexecute RSP.

RSP/EDT MUST BE LOADED TO USE/BULLETIN *BEM*

The BULLETIN READ or WRITE commands can only be issued while EDT or RSP is loaded (use @SY BULLETIN. . .) since there is no workspace unless one of these is active.

SAME MATRIX APPEARS ON BOTH SIDES OF EQUAL SIGN *BASIC*

The same matrix may be referenced on both sides of the equal sign in a MAT statement, a new matrix must be generated.

SAT ERROR INITIALIZING FILE *LIBRARY*

The INIT=YES option has been selected in the command, and the file could not be initialized. This could be due to a hardware error, or an attempt to initialize a non-SAT file.

SCRATCH AREA IS FULL. TEXT NOT ADDED *WORKSPACE*

The program you are using has tried to acquire an additional unit of disk space and could not do so. The last image entered has been lost. Wait and retry or contact the system operator.

SCRATCH ERROR *BEM*

The file specified in the command could not be scratched. This may be a result of an error status being returned from the supervisor, or could have been caused by an attempt to scratch a file which should not be scratched (a system file for example).

SCREEN COMMAND FORMAT ERROR *BEM*

The SCREEN command has been entered incorrectly. Valid options are:

/SCREEN [ROLL / NOROLL] [COP / NOCOP] [,*height* × *width*]

The default is /SCREEN NOROLL,NOCOP,24×80.

SCREEN DIMENSIONS ARE INVALID FOR RSP *RSP*

RSP may only be used with UNISCOPE terminals; the only valid sizes for these terminals are 12 × 80, 16 × 64, 24 × 80, and 24 × 64. These are the only sizes which will be accepted.

SEARCH STRING NOT FOUND *RSP*

RSP has searched the workspace from the current location to the end, but could not find the string requested. Informational message only.

SEARCH STRING NOT FOUND IN LINE-RANGE *EDT*

The Editor has scanned all lines that the user's command has instructed it to, but did not find the string for which it was searching. This is caused by looking for a word or string which is not in the text. Informational message only.

SECOND DEFINITION OF AN ARRAY NOT ALLOWED *BASIC*

Two-dimension statements have been used to define the same variable. Remove one of the statements and rerun.

SECOND DEFINITION OF THE SAME FUNCTION *BASIC*

The same function has been defined twice within the program. Remove one definition and correct the program. Rerun.

SECOND DEFINITION OF SUB — DEFINITION IGNORED *BASIC*

Two subprograms with the same name have been encountered during the compilation process. The second subprogram will be ignored. The second subprogram may have been found in a library element as a result of a library search. This is a nonfatal error.

SEQUENCE PARAMETER ERROR *EDT*

The name or number used to sequence a module is not correct. This may be caused by using more than 16 numeric characters in the name or increment number. See Section 3.1.2.8 of UA-0141.

SET MARGIN FOR DMS FILE NOT AT RECORD 0 *BASIC*

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.

SIMPLE VARIABLE INCONSISTENT WITH CALL *BASIC*

The CALL and SUB lines differ in the specification of a simple variable to be passed to the subprogram. Resolve the inconsistency and rerun.

SOFTWARE CHECK AT ee LLLLLL *BEM*

A software check has been detected by the Monitor. Please take a dump as soon as possible; save all relevant data, and consult your Sperry Univac representative.

SPECIFIED LINE NOT IN FILE *EDT*

The line specified in a replacement expression does not exist in the EDT work space.

SPOOL FILE NOT FOUND — COMMAND IGNORED *RSP*

The file described is not present in the Spool file. Check the spelling of the entries, and check that the correct queue name was specified. The spool element may not have been created yet.

SPOOL I/O ERROR R*U* *RSP*

An I/O error occurred while accessing the system Spool file. Respond R to retry; U to terminate the command. If invalid data has been retrieved, clear the workspace (CLEAR) and retrieve the file again.

SPOOL I/O ERROR WHILE ENTERING TASK *BEM*

The ENTER function has encountered a Spool file access error while writing the command element to the Spool file. If the error persists, contact the system administrator.

SQUARE ROOT OF A NEGATIVE NUMBER UNDEFINED *BASIC*

The SQR function has encountered a negative argument. The square root of a negative number is undefined, thus execution is cancelled.

START AND INCREMENT WILL EXCEED 99999 *BASIC*

The starting number and increment used with the RESEQUENCE command cannot be used as they are, because they would cause one of the new line numbers to exceed the maximum line number (99999) for OS/3 BASIC. Use a different start or increment and reissue the command.

STATEMENT FOLLOWING END/SUBEND NOT SUB/REM *BASIC*

The only permissible statements following an END statement are a REM statement or a SUB statement. Correct the program and rerun.

STATEMENT LABEL NOT IN FILE *EDT*

The label specified on the @GOTO statement could not be found in the EDT proc space.

STATUS COMMAND PARAMETER ERROR *BEM*

The operand of a STATUS command is incorrect. Allowable status commands are:

```
/STATUS          TERM
/STATUS          RESOURCE
/STATUS
```

STOPPED AT xxxxx *BASIC*

A STOP statement has been encountered or an error detected at the line number given by xxxxx.

STRING EXCEEDS 4095 CHARACTERS *BASIC*

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.

SUB: FNX PRECEDES "CALL" *BASIC*

A SUB statement declaring a passed function cannot occur before the statement that calls it (and defines the function parameters). Relocate the subprogram so it occurs after at least one statement that calls it.

SUB NAME IS GREATER THAN 8 CHARACTERS *BASIC*

The name used on a CALL or SUB statement for a subprogram must be a string constant which is not longer than 8 characters. Correct the spelling of the name or shorten its length.

"SUBEND" OR "SUBEXIT" NOT IN A SUB *BASIC*

A SUBEND or SUBEXIT was encountered which was not in a subprogram. The SUBEND must be the last statement in a subprogram.

"SUBEXIT" NOT ALLOWED IN FUNCTION DEFINITION *BASIC*

A SUBEXIT statement was encountered within a multiline user function definition. It can only be issued from the subprogram level.

SUBROUTINE CALLING ITSELF *BASIC*

A CALL statement has been found which references the subprogram in which it resides. Recursive calls in any form are prohibited.

SUBROUTINE LIMIT OF 30 EXCEEDED *BASIC*

BASIC will not accept more than 30 subprograms. Combine several subprograms or change program logic to eliminate a few.

SYSTEM CLOSED TO NEW USERS, TRY LATER *BEM*

The computer operator has closed the system so that no new users will be allowed on. Wait until later to LOGON.

SYSTEM COMMAND NOT RECOGNIZED *BEM*

A command was entered in monitor mode which was not recognized. All commands must begin with one slash and only commands listed below are allowable:

/DELETE *file-info*
/DISPLAY JOBS
/DISPLAY VOLUMES
/EXEC *program*
/FSTATUS *file-info*
/HELP
/INTR
/LOGOFF
/PAUSE *comment*
/PRINT *file-info*
/PUNCH *file-info*
/RUN *program*
/RESUME
/SCREEN
/STATUS RESOURCE
/STATUS TERM
/TYPE *comment*
/VTOC *volume*

"TAB" CANNOT BE USED WITH "PRINT USING" *BASIC*

The TAB function cannot be used while PRINT USING is active. The TAB should be removed, or a semicolon placed before the function call to terminate the USING clause.

TANGENT/COTANGENT OUT OF RANGE *BASIC*

The result of a TAN or COT function evaluation caused an overflow condition. Machine infinity is supplied and execution continues.

TASK ENTERED IN BACKGROUND MODE *BEM*

This is a confirmational message indicating that the Enter file was successfully queued for execution. The task may already have begun, or may be delayed until a batch processor becomes available.

TERMINAL ALREADY LOGGED ON, PROCEED *BEM*

The previous user did not LOGOFF; this terminal is still logged on.

TERMINAL IDLE TOO LONG, REPLY OR BE CANCELLED *BEM*

This terminal has had no activity for a long period of time, and is assumed to have been left idle. If this terminal is still in use, reply within 30 seconds, or BEM will log the terminal off. The time limit before this message is displayed is set by the system administrator.

THE SUBROUTINE DEFINED IS NOT REFERENCED *BASIC*

This is an informational message only. It notifies the user that he has included a subprogram (either implicitly via LIBRARY or explicitly in the workspace) which is never called. It should be eliminated as it only takes up memory. Compilation continues.

TIME UP — PROGRAM LOOPING *BASIC*

The time limit specified in the TIME statement has been exceeded by the program. It may be looping, or it may require that the time limit be increased.

TOO MANY TAB STOPS *EDT*

More than eight tab stops have been used with the TAB command. See Section 3.2.3 in UA-0141.

TYPE OF FUNCTION PARAMS INCONSISTENT IN CALL *BASIC*

The functions passed to a subprogram do not agree in type or number of parameters expected. Check the CALL statements to see that any functions passed contain the same number and type of parameters, then check the subprogram to be sure it references the function correctly.

UNABLE TO CREATE SPOOL FILE *RSP*

RSP could not successfully build the desired Spool file. Check parameters and retry. If the problem persists, consult your Sperry Univac representative.

UNCORRECTED ERROR IN SOURCE *BASIC*

One of the statements flagged during the previous OLD command has not been eliminated or corrected. The number of that line is shown.

UNKNOWN ERROR ON SAT FILE *LIBRARY*

BEM has received an error code from the SAT processor which it does not expect. If the command issued does not violate any of the constraints placed on it by BEM, contact your local Sperry Univac representative.

UNRECOGNIZABLE COMMAND *EDT*

A command keyword has been used which the Editor does not recognize. Check all keywords used against Appendix A of UA-0141.

USER DID NOT SUPPLY FILE NAME & NO DEFAULT GIVEN *FILE/LIBRARY*

The user has issued a command which requires that a file-name be specified; no filename was stated in the command. If the user expected to use a default file specification, he should contact the system administrator, as the administrator did not declare a default file for this account. To correct the command, enter a filename explicitly.

USER LOGGED OFF, CANCELLED BY OPERATOR *BEM*

The operator has cancelled your task for some reason. Contact the operator to find out why.

USER LOGGED OFF, END OF FILE ON CONTROL STREAM *BEM*

This message is only issued by a batch processor. It indicates that the processor attempted to read more cards from the enter stream and found none left. The enter task is automatically logged off. This is usually caused by a misinterpreted or mistyped LOGOFF command.

USER LOGGED OFF, NO RESPONSE IS ALLOTTED TIME *BEM*

This message is issued 30 seconds after the "TERMINAL IDLE TOO LONG. . ." message if no response is made. To use this terminal, the next user need only log on again.

USER LOGGED OFF, TERMINAL IS NOW FREE *BEM*

This indicates successful completion of a LOGOFF command.

USER'S ACCOUNT DOES NOT PERMIT WRITING TO FILES *FILE/LIBRARY*

The user has attempted to write or update a file and is not permitted to by his account description. To remove this restriction, contact the system administrator to change the access permission.

USER'S ACCOUNT PROHIBITS ACCESS TO THIS FILE *FILE/LIBRARY*

The account description for this user does not permit the specified file to be accessed. This usually is a result of accessing a file other than the default file if only that file is permitted. To remove this restriction, contact the system administrator.

VOLUME IS NOT AVAILABLE TO THE BEM SYSTEM *LIBRARY/FILE*

The disk volume you have requested is mounted, but has not been made available to the BEM system by the system administrator. Contact the system administrator to have the pack included.

VOLUME NAME IS NOT SPECIFIED *LIBRARY/FILE*

The user has not supplied the name of the disk volume to scan, and the volume name is not in the catalog.

VOLUME NOT MOUNTED ON A DISK DRIVE *LIBRARY/FILE*

The volume requested is not mounted on a disk drive. Contact the operator to have the volume mounted.

WAITING FOR OPEN FILE TABLE ENTRY *LIBRARY/FILE*

An "Open File Table Entry" in the preamble could not be secured for this file access. BEM will wait until another file access terminates and releases its entry. If the user does not wish to wait, the interrupt facility of BEM may be used to terminate the file access. If this problem occurs frequently, contact the system administrator to have more entries placed in the preamble.

 # OF FUNCTION PARAMS INCONSISTENT IN CALL *BASIC*

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.

#0 INVALID ON CHAIN *BASIC*

Channel zero, the terminal, may not be used as the file from which the chained program is to be read. A data management, temporary, or library file must be used.

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Application Services for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

_____
(Document Title)

_____ _____ _____
(UA No.) *(Revision No.)* *(Update No.)*

Comments:

Cut along line.

From:

_____
(Name of User)

_____
(Business Address)

Fold on dotted lines, staple, and mail. (No postage stamp necessary if mailed in U. S. A.)

Thank you for your cooperation.

Staple

Staple

Fold

FIRST CLASS
Permit No. 21
Blue Bell, Pa.

BUSINESS REPLY MAIL — no postage necessary if mailed in the United States

Postage will be paid by

SPERRY UNIVAC

Attn: Manager, Application Services

P. O. Box 500

Blue Bell, PA 19422

Fold

Cut along line.