GC20-1803-1

Systems

IBM Virtual Machine Facility/370: BASIC Language Reference Manual

Release 1 PLC 5

This publication describes the BASIC language facility of the IBM Virtual Machine Facility/370 (VM/370). It includes a precise description of the language as well as a guide to creating and running BASIC programs under the Conversational Monitor System (CMS) of VM/370.

IBM

Second Edition (April 1973)

This edition is a major revision of,and makes obsolete, GC20-1803-0. The document has been reorganized to allow easy access to CP/CMS usage information, BASIC language structure and elements, and reference information. Changes herein also reflect changes and enhancements of the BASIC language processor.

This edition corresponds to Release 1 PLC 5 (Program Level Change) of the IBM Virtual Machine Facility/370 and to all subsequent modifications until otherwise indicated in new editions or Technical Newsletters.

Changes are periodically made to the specifications herein; before using this publication in connection with the operation of IBM system, refer to the IBM System/360 and System/370 Bibliography, GA22-6822, and the IBM System/370 Advanced Function Bibliography, GC20-1763, for the editions that are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address comments to: IBM Corporation, VM/370 Publications, 24 New England Executive Park, Burlington, Massachusetts 01803. Comments become the property of IBM.

PREFACE

This manual serves as a reference guide to the BASIC language facility of VM/370. This BASIC language facility consists of the CALL-OS BASIC (Version 1.2) Compiler and Execution Package and, therefore, has complete programming language compatability with CALL-OS BASIC Version 1.2. The material is organized to provide a concise definition and syntactical reference to the various elements of the language. Examples are provided.

Major subjects include program structure, elements of statements, input and output, program statements, and the CMS BASIC command. Program limits and error messages are covered in the appendixes.

Although some operating information is included, this book is not a complete description of the CMS facilities which will be used by the BASIC programmers. It contains general information needed when writing a VM/370 BASIC program.

Prerequisite Publications

Because the VM/370 BASIC user enters his programs and data using the CMS environment, three manuals are required for effective use of VM/370 BASIC:

IBM Virtual Machine Facility/370:Command Language User's Guide, Order No. GC20-1804

IBM Virtual Machine Facility/370:EDIT Guide, Order No. GC20-1805

IBM Virtual Machine Facility/370:Terminal User's Guide, Order No. GC20-1810

Corequisite Publications

IBM Virtual Machine Facility/370: Command Language User's Guide, Order No. GC20-1804

VM/370 BASIC Reference Summary, GX20-1924

In this publication, the term "3330 series" is used in reference to both the IBM 3330 Models 1 and 2 Disk Storage Facility and the IBM 3333 Model 1 Disk Storage and Control.

Information in this publication (if any) about the CMS Batch facility or for the IBM System/370 Models 165 II and 168 is for planning purposes only.

Terminal Equivalence

Terminals which are equivalent to those explicitly supported may also function satisfactorily. The customer is responsible for establishing equivalency. IBM assumes no responsibility for the impact that any changes to IBM-supplied products or programs may have on such terminals.

Summary of Amendments for GC20-1803-1 VM/370 Release 1 PLC 5

Addition of the following VM/370 Programming Functions

Specification Change

- The Virtual =Real Performance Option
- The Dedicated Channel Performance Option
- The Virtual and Real Channel-to-Channel Adapter

Support for the following devices

Specification Change

- The IBM 3211 Printer
- The IBM 3410/3411 Magnetic Tape Subsystem
- The IBM 3330 Disk Storage Model 2
- The IBM System/370 Models 155 II and 158

Availability of Programs that Execute under CMS

New Program

The IBM Program Product PL/I Optimizing Compiler is now available.

Miscellaneous Changes

Maintenance: Documentation Only

Additions, deletions, and corrections too numerous to list are included in this revision. Generally, the document has been rewritten and reorganized to allow easy access to CP/CMS usage information, BASIC language structure and elements, and reference information.

CONTENTS

The BASIC language is a high level programming language designed to be used in an interactive environment. BASIC is, as its name implies, a language that can be learned quickly and used conveniently in both scientific and commercial applications.

BASIC provides facilities for evaluation of ordinary algebraic expressions containing various types of constants and variables, a variety of input/output methods, and many intrinsic (built-in) functions. It is powerful problem-solving tool when used under the IBM Virtual Machine Facility/370 (VM/370) Conversational Monitor System (CMS).

For ease of use, this manual is divided in three parts. Part 1 tells how to use the BASIC language in the VM/370 CMS environment. Part 2 describes the elements of the language, that is, the structure and facilities of the language. Part 3 is reference information: A chart summary of all the statements of the language and their uses, and an alphabetical listing of all of the statements of the language with their formats and usage rules.

SYNTAX CONVENTIONS

The following conventions are used in this manual to describe the formats of VM/370 BASIC statements:

- Uppercase letters, digits, and special characters must appear exactly as shown.
- Information in lowercase letters must be supplied by the user.
- Information contained within braces { } represents alternatives, one of which must be chosen.
- Information contained within brackets [ ] represents an option that the user can omit.
- An ellipsis (a series of three periods) indicates that a variable number of items may be included in a list. A list whose length is variable is specified by the format x(1), x(2), x(3), ... x(n) indicating that from 1 to n entries may appear in the list.
- The appearance of one or more items in sequence indicates that the items, or their replacements, should also appear in the specified order.
- A vertical bar | indicates that a choice must be made between the item to the left of the bar and the item to the right of the bar.

Thus , the format description

MAT GET [u:|f,]m_1 [(d_{11} [,d_{12}])],m_2 [(d_{21} [,d_{22}])],...,m_n [(d_{n^1} [,d_{n^2}])]

indicates that:

- MAT GET, the colon or comma, and succeeding commas and parentheses must appear as needed;
- The user may enter a value for the variable represented by u or f, and for one or more m(d,d) specifications;
- Either a u or f value (but not both) may be specified, and neither is required;
- Use of (d,d) with each m specification, and of the second d within each (d,d) is also optional.

PART 1: PROGRAMMING IN VM/370 BASIC

Part 1 of this manual describes the VM/370 environment in which BASIC programs may be coded.

This part is meant for use by those programmers who are not familiar with the interactive terminal environment provided via CMS. It is, therefore, brief and contains only enough information to use the VM/370 BASIC language processor. All of CMS facilities are described in the IBM Virtual Machine Facility/370: Command Language User's Guide, Order No. GC20-1804.

The VM/370 BASIC language processor runs under the VM/370 Control Program (CP) and the Conversational Monitor System (CMS). CP is the control program that controls the VM/370 system resources. CMS is an operating system that provides a comprehensive set of conversational facilities to a single user, among them the VM/370 BASIC language processor.

CP AND CMS FOR BASIC USERS

The VM/370 BASIC programmer uses the facilities of both CP and CMS when he enters his BASIC program.

The CP facilities are accessed by issuing the LOGIN command. This command connects the terminal to VM/370, which permits access to CMS. It takes the form:

```
login userid
```

where *userid* is the user identification code that identifies the user to the VM/370 system.

If the userid is recognized, the system responds by asking the user for his password.

After the userid and password are entered correctly, the user can access CMS by means of the IPL command, which takes the form:

```
ipl sysname
```

where *sysname* is the name of some VM/370 operating system. In this case, the user types 'ipl cms', which loads his CMS virtual machine.

Once in the CMS environment, the user can begin coding his BASIC program using the facilities of the CMS EDIT command. When the program is complete, it can be executed using the CMS BASIC command.

The sample terminal session at the end of this section shows how some of the CP and CMS commands are used.

CMS FILES

In VM/370, a collection of data is called a file. The rules for creating and naming files can be found in the publication IBM Virtual Machine Facility/370: Command Language User's Guide, Order No. GC20-1804.

Files written to be executed by the BASIC language processor must have a filetype of BASIC. These BASIC files are created using the CMS Editor, that is, written in CMS EDIT mode.

THE CMS EDITOR

The CMS Editor provides the environment required for writing BASIC programs. To use the EDIT command to write a BASIC program, the user can type:

```
edit filename basic
```

where filename is the name of the program or data file being created.

All of the EDIT subcommands are available for creation and maintenance of files (for example, INPUT, CHANGE, IMAGE, TYPE, SAVE, and QUIT). Other EDIT subcommands are also available for use; see the publication IBM Virtual Machine Facility/370: EDIT Guide, Order No. GC20-1805, for a complete description of the EDIT facilities.

When the BASIC program is complete, it must be saved using the EDIT subcommand FILE before it can be executed. The FILE subcommand is entered as follows:

```
file
```

FILE causes the program or data file to be stored on the user's primary disk.

THE BASIC COMMAND

Once the program has been filed, it can be executed using the CMS BASIC command. The BASIC command invokes the BASIC language processor and takes the form:

```
basic filename          [ (LONG) ]
```

where filename specifies the name of the file to be compiled and executed. The file must have a filetype of BASIC and contain fixed-length records of up to 256 characters.

(LONG) is the option for long-form precision numbers. If not specified, short form is assumed.

A SAMPLE TERMINAL SESSION

The following example represents a terminal session for coding a BASIC program using the facilities of VM/370.

When the user switches on his terminal and hits the Attention key, VM/370 responds and the terminal session begins.

```
vm/370 online
(press attention)
login user1
ENTER PASSWORD:

READY MESSAGE - - - - -

ipl cms
CMS 02/02/73 FRI 08.12.38
edit prog1 basic
NEW FILE:
EDIT:
input
INPUT:
  10    .
        .
        .
        BASIC
        PROGRAM
        .
        .
  60    .
(press return)
EDIT:
file
R;
```

```
basic prog1
        .
        .
        .
     PROGRAM
     OUTPUT
        .
        .
        .
R;
```

REFERENCES

The BASIC programmer is encouraged to reference the following two publications to learn more about the functional capabilities of VM/370. The complete EDIT facility is described in IBM Virtual Machine Facility/370: EDIT Guide, Order No. GC20-1805. All other commands are explained in the IBM Virtual Machine Facility/370: Command Language User's Guide, Order No. GC20-1804. Some of the commands most frequently used by the BASIC programmer are listed below:

Command	Action
ACCESS	Sets up a device to write and/or read
EDIT	Enters and modifies a file
ERASE	Deletes a file from a read/write disk
LISTFILE	Types file statistics to the terminal
LINK	Attaches a device to a virtual machine
MOVEFILE	Moves files from one device to another
PRINT	Writes a file to the printer
PUNCH	Punches a file to the card punch
QUERY	Obtains data about system characteristics
READCARD	Reads a file from the virtual card reader
RELEASE	Frees up a device from the current read/write configuration
RENAME	Changes the name of a file
SET	Establishes system characteristics
TYPE	Lists a file at the terminal

PART 2: THE VM/370 PROGRAM STRUCTURE AND LANGUAGE ELEMENTS

This part of the manual is in two sections: a description of the structure of a VM/370 BASIC program and a description of the BASIC language elements.

BASIC PROGRAM STRUCTURE

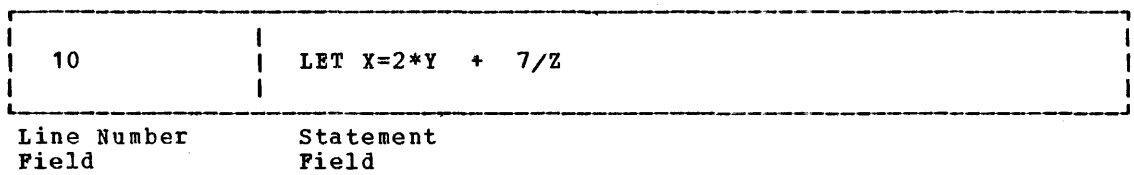
This section describes the way in which the BASIC program is coded under VM/370 CMS.

The procedure for coding a program is begun by using the CMS EDIT command to create a new file. Once the file is created, the coding can begin by typing in statement lines at the terminal.

Structurally, the statement line is coded in two fields. Logically, the statement can be either executable or non-executable.

THE BASIC STATEMENT LINE

The BASIC statement line is structured in two fields: the line number field and the statement field, as shown below:



THE LINE NUMBER FIELD

Each BASIC program statement must be preceded by a line number. If the program is being coded in the EDIT INPUT mode, the line numbers are supplied automatically.

Line numbers may be entered in any numeric sequence; the processor sequences all the statements before it executes a program.

The line numbers may be up to five characters long. The line number may not contain embedded blanks.

Throughout this text, the term *line number* is used to identify the number preceding each statement. This term is synonymous with *statement number* as used in other discussions of the BASIC language.

In all of the discussions of statements in this publication, the line number is understood as a part of the line and is therefore not used in the format specification.

THE STATEMENT FIELD

The statement field contains the actual BASIC statement being entered.

This field must be separated from the line number field by a blank. If the program is in the EDIT INPUT mode, the line number and the required space are printed automatically.

The statement field is terminated when the user hits the return key or when he uses the CMS logical line end symbol (the commercial pound sign #).

USE OF BLANKS AND COMMAS

Blanks are, in general, ignored by the BASIC language processor. They are inserted mainly to improve program readability.

However, in certain cases blanks are required. Blanks are required to separate the line number and the BASIC statement. Also, in a statement that requires operands, blanks must be entered to separate the operation from the operands.

Commas are used to separate operands.

THE BASIC SOURCE STATEMENT LINE

The source statement line is a line that contains a complete BASIC statement. A source statement line is terminated when the programmer hits the return key or when he enters the CMS logical line end symbol (the commercial pound sign #).

There is no means for continuing a source statement line in BASIC.

A source statement line may contain only one statement.

EXECUTABLE AND NON-EXECUTABLE STATEMENTS

Logically, BASIC statements can be either executable or non-executable. An executable statement specifies a program action, for example X=5. A non-executable statement provides information necessary for program execution, for example DATA 1, 2.5, 6E-7.

Executable and non-executable statements may be mixed.

Transfer of control to a non-executable statement causes control to pass to the next executable statement.

This section includes descriptions of the BASIC language elements. The major topics are: symbols used in the language, the functions defined as part of the language, how to write expressions in BASIC, Input/Output, and program chaining.

VM/370 BASIC CHARACTER SET

A BASIC program is written for use under the CMS subsystem of VM/370 using the following character set:

1. Letters: 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 @ # $

2. Digits: 0 1 2 3 4 5 6 7 8 9

3. Special characters:

'	Single quote	*	Asterisk (Multiplication)
"	Double quote	/	Right oblique (Slash) (Division)
<	Less than	**	Exponentiation
<=	Less than or equal to	(	Left parenthesis
=	Equal to	)	Right parenthesis
=>	Greater than or equal to	!	Exclamation mark
>	Greater than	,	Comma
<>	Not equal to	.	Period
&	Ampersand	;	Semicolon
+	Plus	:	Colon
-	Minus		Blank
		|	Vertical bar

Any valid terminal character not listed is a non-BASIC character and may be used only where specifically noted.

COMMENTS

Comments may be coded anywhere in a BASIC program using the REM statement, for example:

```
10 REM THIS IS AN EXAMPLE STATEMENT
```

If execution is directed to the line number of a REM statement, control passes to the next statement in sequence.

SYMBOLS USED IN BASIC

There are two types of symbols in BASIC: constants and variables. A constant is a symbol whose value does not change during program execution. A variable is a symbol whose value may change during program execution.

This section describes the various types of BASIC constants and variables.

WRITING CONSTANTS IN BASIC

There are three types of constants in VM/370 BASIC: numeric constants, internal constants, and literal constants.

Numeric Constants

There are three types of numeric constants: integer, fixed-point, and floating-point. Also, there are two basic forms in which formatted numeric constants can be entered in a program: short-form and long-form.

Decimal numbers are printed in either fixed-point form (F format) or floating-point form (E format).

SHORT-FORM INTEGER NUMBERS: An integer format (I format) is used to print integer values. Up to eight decimal digits may be printed for each short-form integer whose absolute value is less than 16777216. For example:

```
17
203167
5
9993456
```

SHORT-FORM INTEGER NUMBERS: An integer format (I format) is used to print integer values. Up to eight decimal digits may be printed for each short-form integer whose absolute value is less than 16777216. For example:

```
1.2076
+783347
-.003424
```

SHORT-FORM FLOATING-POINT NUMBERS: The short-form floating-point format (E format) is used to specify numbers whose magnitude is less than 10^{-6} or greater than 10^{7}. The number takes the form:

```
[±]d.ddddE±ee
```

where d specifies a digit
e specifies an exponent

E Format	F Format	I Format	Equivalent Number
-1.70834E+02	-170.834	-171	-170.834
5.43311E-05	+.000054	+0	+.000054331
2.17787E+00	+2.17787	+2	+2.17787
-6.72136E-02	-.067214	+0	-.0672136
9.68E-07	.000000	+0	+.000000968

If the exponential notation (E format) is used, the value of the constant is equal to the number on the left of the E multiplied by 10 raised to the power of the number following the E. The magnitude of a numeric constant must be less than 1E+75 and greater than 1E-78.

LONG-FORM NUMBERS: An I format is used to print an integer value up to 15 digits in length whose absolute value is less than 10^{15}, using either the PRINT or the PRINT USING statement.

Decimal numbers written in F format are used to print decimal values of up to 15 digits with decimal point. F format long-form numbers may be printed using only the PRINT USING or MAT PRINT USING statement.

Decimal numbers written in E format are printed using either the PRINT, PRINT USING, or MAT PRINT USING statement.

The PRINT statement is used to print a value with a sign, a decimal point, up to ten decimal digits, the letter E and a signed exponent. With the PRINT USING statement, a value having a sign, a decimal point, up to 15 decimal digits, an E, and a signed exponent can be printed.

PRINT FORMAT: The print format that applies when a PRINT statement is used depends on the value to be printed. I, F, or E format will be selected as follows.

	BASIC (Short-Form)	BASICL (Long-Form)
I format	x an integer, $\|x\| < 16777216$	x an integer, $\|x\| < 1E15$
F format	x noninteger, $.1 \leq \|x\| < 1E6$	none
E format	other numeric values	other numeric values

Alternatively, the user may enter PRINT USING or MAT PRINT USING statements to format print lines.

ADDITIONAL CONSIDERATIONS: It is not possible to exactly represent all noninteger decimal numbers in floating-point form. VM/370 BASIC uses a "truncation-in/rounding-out" algorithm for decimal/floating-point conversion. That is, when an input value is converted to floating-point form, excess fractional digits are dropped so that the value to be stored fits in either one word or two words (depending on whether short-form or long-form arithmetic is to be used).

When the floating-point number is reconverted to decimal for output, it is rounded up, to compensate for any fractional value discarded earlier.

For most applications, decimal/floating-point conversion will cause no problems. However, the difference between decimal numbers and their

floating-point approximations as accumulated during a series of calulations may be significant.

This effect can also be observed when using the INT function.

The user can provide for calculation differences by any of several methods. For example, he can:

1. Try running his program in BASICL to obtain greater precision.
2. Force rounding on input by adding a very small decimal fraction to each input value.
3. Use multiplication to scale all numbers to integers prior to calculation and then divide to position the decimal point for output.

Assume that a VM/370 BASIC program has been written to accept a decimal value as input from the terminal and add that number to itself repetitively. Such a program is shown below.

```
type test basic
10 INPUT A
20 FOR I=1TO11
30 B=B+A
40 NEXT I
50 PRINT B
60 END
```

Now assume that the decimal fraction .3 is provided as input. If the program is executed using short-form floating-point arithmetic, results are as shown below.

```
basic test


?.3
 3.29999

R;
```

To avoid this truncation error, any of the approaches described above can be used, as shown by the following examples.

(1) Try running the program in long-form precision

```
basic test (long)

?.3
 3.300000000E+00

R;
```

(2) Force rounding on input

```
basic test

?.300001
 3.3

R;
```

(3) Scale numbers to integers for calculation (in this case, by modifying program statements before running the program)

```
15 A=10*A

50 PRINT B/10

basic test

?.3
 3.3

R;
```

Internal Constants

Three internal constants are provided in VM/370 BASIC. They represent Pi, e, and the square root of 2. The names of the internal constants may be used in calculations where the values of the constants are needed. They are called &PI, &E, and &SQR2. The values inserted by the system are:

Name	Short-form Value	Long-form Value
&PI	3.141593	3.141592653589793
&E	2.718282	2.718281828459045
&SQR2	1.414214	1.414213562373095

For example:

```
10 LET X=&PI*Y(2)
20 LET R=&E+4*Z**3
30 LET Y=&SQR2*C**4
```

Literal Constants

A literal constant is a character string enclosed by a pair of single or double quotation marks. The two general forms of a literal constant are:

```
"[c...]"

'[c...]'
```

where c is any character.

A single quote may appear in a character string bounded by double quotes, and a double quote may appear in a character string bounded by single quotes. However, when a character string contains the boundary character, it must be identified by two consecutive boundary characters.

The following examples illustrate how character strings may be represented as literal strings:

Character String	Literal String
ABCD	"ABCD" or 'ABCD'
ABC'D	"ABC'D" or 'ABC''D'
ABC"D	"ABC""D" or 'ABC"D'

WRITING VARIABLES IN BASIC

A variable is a symbol whose value may change during the execution of a program. In BASIC, there are two general types of variables: simple variables and array variables.

Simple Variables

There are two types of simple variables: simple numeric variables and simple alphameric variables, as explained below.

SIMPLE NUMERIC VARIABLE: A simple numeric variable is named by a letter (or a character from the extended alphabet) or a letter followed by a digit. Examples are:

A, B1, @, #4, $9

A simple numeric variable can be assigned only a numeric value; the initial value of all simple numeric variables is zero.

SIMPLE ALPHAMERIC VARIABLE: A simple alphameric variable is named by a letter followed by the character $. Examples are:

A$, B$, X$

A simple alphameric variable can contain only an 18-character literal value; the initial value of all simple alphameric variables is 18 blank characters.

Arrays and Array Variables

An array is an ordered set of data members which may be one-dimensional or two-dimensional.

There are two types of arrays in BASIC: alphameric arrays and matrixes (also called numeric arrays).

Arrays are named by means of array variables. Alphameric arrays are named with a single letter followed by a $. Matrixes are named using single letters.

REFERENCING ARRAY MEMBERS: An array member is referenced using a subscript array name in the form:

```
arrayname(dim1[,dim2])
```

where arrayname is the name of the array being referenced.

dim1 is the dimension specification for row number.

dim2 is the dimension specification for column number.

SUBSCRIPT EVALUATION: Subscripts (dimi) are expressions evaluated in floating-point arithmetic and then truncated to an integer. For instance, 3.61727E+00 would be truncated to the integer value 3.

The number of subscripts used to reference an array member must equal the number of dimensions for the array.

The maximum value of the subscript number must be within the bounds defined for the array.

MATRIXES: A matrix is a numeric array named using a single letter. There may be as many as 29 arrays in a VM/370 BASIC program.

A matrix may have one or two dimensions.

A matrix may contain only numeric values; the initial value of each numeric array number is zero. Examples are:

```
 40 GET I(10,12)
 60 LET J(I)=A(K)*R
100 PUT L(*)
```

For more information on matrixes, see the section "Matrix Manipulation," below.

ALPHAMERIC ARRAYS: Alphameric arrays are named by a single letter followed by a dollar sign and may have only one dimension.

Alphameric arrays contain 18-character members whose values are literal constants; the initial value of each alphameric array member is 18 blank characters. Examples are:

```
 40 GET I$(10)
 60 LET J$(I)=A$(K)
100 MAT A = B + C
140 MAT F = INV(G)
```

DECLARING ARRAYS: Array declaration is the process of allocating an array to a user program. The name and dimensions of the array are defined in the declaration.

Arrays may be defined explicitly in the DIM statement or implicitly by their appearance in a program. See the Reference Information Part of this manual for rules on how to use the DIM statement.

An array is implicitly declared by the first reference to one of its members (for exception, see MAT instructions), if the specified array has previously not been defined by a DIM statement. The array is declared to have one dimension (10) when a member is referenced by an array variable with one subscript. The array is declared to have two dimensions (10,10) when a member is referenced by an array variable with two subscripts.

Array dimensions and referencing start at one. That is, an array having one dimension (n) has n members, and an array having two dimensions (m,n) has m times n members, where m specifies the number of rows and n, the number of columns.

The maximum storage capacity for arrays is 28,668 bytes. The maximum number of members in various types of arrays is shown below.

	Short-Form	Long-Form	Alphameric
Number of bytes per array member	4	8	18
Maximum number of array members	7167	3583	1592

MATRIX MANIPULATION

A matrix is a system of values arranged in a one-dimensional or two-dimensional numeric array.

The limits of a matrix must be explicitly defined by a DIM statement, or implicitly defined by its appearance in a program, before the matrix is used in any MAT operations. A matrix may then be redimensioned by appending the new dimension (enclosed in parentheses) or dimensions (enclosed in parentheses and separated by a comma) to any of the following matrix statements:

Matrix CON function

Matrix IDN function

Matrix ZER function

MAT GET

MAT READ

MAT INPUT

When a matrix is two-dimensional, the first dimension defines the number of rows, and the second dimension defines the number of columns.

Implicit Matrix Declaration

A matrix can be implicitly defined in a program by the use of a variable name followed by two subscripts enclosed in parentheses. Implicitly defined matrixes cannot be used in MAT operations.

Redimensioning A Matrix

Redimensioning of a matrix is the process of adjusting the contents of a matrix into a new pattern of rows and columns. For example, adjusting a 5 by 10 matrix into a 2 by 25 matrix.

Redimensioning can neither increase the total size of a matrix nor change the number of dimensions of the matrix. If redimensioning causes the number of matrix elements as originally declared to be exceeded, program execution is terminated.

Currently defined dimensions are observed when executing a matrix statement. Redimensioning occurs before the operation specified in the statement containing the new dimensions.

The following example shows how redimensioning occurs:

```
120 DIM A(20,40)
130 DIM B(15,100)
.
.
.
250 MAT READ A(10,40)
260 MAT READ B(1,15)
.
.
.
```

Matrix A is originally a 20 x 40 matrix. Line 250 redefines the limits to 10 x 40. Similarly, matrix B is redefined from a 15 x 100 matrix to a 1 x 15 matrix.

Arithmetic With Matrixes

The BASIC language allows arithmetic to be performed using matrix elements as operands. There are five arithmetic operations permitted: addition, subtraction, multiplication, scalar multiplication, and assignment.

These operations cause entire matrixes to be added, subtracted, etc. For format and usage rules for using these operations, see the Reference Information part of this manual.

FUNCTIONS IN BASIC

There are three types of functions in VM/370 BASIC: intrinsic functions, user-defined functions, and matrix functions.

INTRINSIC FUNCTIONS

An intrinsic function is one whose meaning is predefined by the language processor. These functions are provided to facilitate the writing of VM/370 BASIC programs. The available functions may be used very much as a variable would be used. For example, let

```
A = SIN(23)
Z = LOG(X) + LOG(Y)
```

The intrinsic functions provided as part of the VM/370 BASIC language are listed below. The allowable limits for arguments passed to these functions are given in "Appendix A: VM/370 BASIC Program Limits."

Function	Allowable Limits
SIN(x)	Sine of x radians
COS(x)	Cosine of x radians
TAN(x)	Tangent of x radians
COT(x)	Cotangent of x radians
SEC(x)	Secant of x radians
CSC(x)	Cosecant of x radians
ASN(x)	Angle (in radians) whose sine is x
ACS(x)	Angle (in radians) whose cosine is x
ATN(x)	Angle (in radians) whose tangent is x
HSN(x)	Hyperbolic sine of x radians
HCS(x)	Hyperbolic cosine of x radians
HTN(x)	Hyperbolic tangent of x radians
DEG(x)	Convert x from radians to degrees
RAD(x)	Convert x from degrees to radians
EXP(x)	Natural exponent of x (e to the power x)
ABS(x)	Absolute value of x (|x|)
LOG(x)	Logarithm of x to the base e (ln x)
LTW(x)	Logarithm of x to the base 2
LGT(x)	Logarithm of x to the base 10
SQR(x)	Positive square root of x
RND(x)	A random number between 0 and 1 (x is a seed, if specified)
INT(x)	Integral part of x
SGN(X)	Sign of x, defined as: SGN(x) = -1 if x < 0 SGN(x) = 0 if x = 0 SGN(x) = +1 if x > 0
DET(A)	Determinant of the square matrix A

Note: In VM/370 BASIC, Version 1.1, the RND function required an argument, but that argument was not used. The same sequence of pseudo random numbers was always generated. With Version 1.2, if an argument is present, it is used as a seed for the pseudo random number generator. To obtain successive numbers in this sequence, thus seeded, RND without an argument should be specified. For example:

```
10 A = RND(6)
   .
   .
   .
20 FOR I = 1 to 1000
21 PRINT RND
22 NEXT I
```

If the first reference to the RND function contains no argument, VM/370 BASIC will supply a random seed. Thus, RND, in Version 1.1 and 1.2, can be summarized as follows.

Version 1.1 RND(x)	Returns successive numbers of the one available sequence of pseudo random numbers.
Version 1.2 RND(x)	Returns the first number of a sequence of pseudo random numbers; the sequence depends on the value of x.
RND	• When this is first reference to RND during this running of the program: Returns the first number of a sequence of pseudo random numbers; the sequence depends on a system-supplied seed.
RND	• When this is not first reference to RND during this running of the program: Returns successive numbers of the sequence of random numbers determined by either means indicated for Version 1.2, above.

This change to the RND function in Version 1.2 is the only instance necessitating user changes to certain types of instruction sequences in existing programs. An instruction sequence such as the following may currently exist in a VM/370 BASIC program:

```
100 A = INT(10*RND(X))
110 IF A = 0 THEN 100
```

This program will loop if compiled under VM/370 BASIC Version 1.2 if the seed, X, is such that A equals zero. The RND function will be reseeded with the same value of X at each execution of line 100, and A will always be zero.

USER-DEFINED FUNCTIONS

A user-defined function is one whose meaning is defined by the user via the DEF statement. The user function is named by the characters FN followed by a letter. For example, FNA(X) could be defined as:

```
10 DEF FNA(X)=2+3*X-5*X**2
```

FNA(X) can then be used in the same manner as an intrinsic function.

Rules for writing and using the DEF statement can be found in the Reference Information part of this manual.

MATRIX FUNCTIONS

There are five functions in VM/370 BASIC for use in manipulating matrixes. These functions are: CON, IDN, INV, TRN, and ZER. For information on how to write these functions and use them, see the Reference Information part of this manual.

VM/370 BASIC OPERATORS

There are three types of operators used in BASIC for the formation of expressions: unary, arithmetic, and relational. The lists below show how these operators are written in VM/370 BASIC.

UNARY OPERATORS

Characters	Meaning
+	the value of
-	the negative value of

ARITHMETIC OPERATORS

Characters	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

RELATIONAL OPERATORS

Character	Meaning
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
=	Equal
<>	Not equal

WRITING EXPRESSIONS IN VM/370 BASIC

An expression consists of one or more numeric variables, numeric constants, internal constants, and functions, together with unary and arithmetic operators. As noted below, parentheses may be included if necessary.

Alphameric variables, literal constants, and relational operators are

not allowed in expressions.

An expression is evaluated by performing the indicated operations as shown below. When these rules are not definitive, operations are performed from left to right in the expression.

1. Operations within parentheses are performed before operations not within parentheses.

2. Operations on the same level are performed in the order in which they appear from left to right in the expression.

   Note: The expression A**B**C is evaluated as (A**B)**C -- not as A**(B**C).

3. Operations are performed in sequence from the highest level to the lowest level. The priorities of operation are:

   a. operations within parentheses

   b. ** (exponentiation)

   c. * or /

   d. + or -

Examples of expressions are:

```
A1
-6.4
SIN(R)
X+Y-Z
X3/(-6)
-(x-(x**2/2)+x**(y*z))
```

This last expression corresponds to the algebraic expression:

```
-(x-(x**2/2)+x**(y*z))
```

Expressions resulting in an imaginary or mathematically undefined value are not evaluated. The system generates an error message appropriate to the function involved and terminates execution (see "Execution Error Messages" in Appendix B). If an arithmetic exception is involved, the system continues program execution after taking the specified action (see "Execution Exception Messages" appearing in "Appendix B: VM/370 BASIC Error Messages").

BASIC INPUT AND OUTPUT

When using VM/370 BASIC in the solution of problems, it is often necessary to be able to manipulate large groups of data. Groups of associated data items are called data files. Methods of entering data files into a BASIC program are:

1. Internal Specification

2. Terminal Input/Output

3. Disk Input/Output

INTERNAL SPECIFICATION

The internal specification statements are READ, DATA, and RESTORE. The use of the DATA statement causes data to be compiled into the program. The data is stored with the program when the program is saved. The READ and RESTORE statements utilize the data defined in DATA statements.

The DATA statement is used to create tables of data values in the program. For example:

```
DATA 100.7, -23.2, 438.8, 201.3, 816.9, 537.8
```

The values listed after the DATA statement can be accessed by the use of a READ statement.

For example:

```
10 DATA 100.7, -23.2, 438.8, 201.3, 816.9, 537.8
20 READ X
30 READ Y,A
40 RESTORE
50 READ Z
```

Line 20 would cause the first value in the data list (100.7), to be stored in the variable X. Line 30 would assign the second value in the list (-23.2) to the variable Y and the third value (438.8) to A. The RESTORE statement in line 40 sets the list pointer back to the beginning of the data list. Therefore, line 50 would assign the first value in the list (100.7) to Z.

DATA statements may appear anywhere in the program. Each time a READ statement is executed, the next sequentially available data value will be assigned to the variable(s) specified in the READ statement. This will continue until the data list or lists are exhausted, or until a RESTORE statement is executed.

TERMINAL INPUT/OUTPUT

The statements used for terminal input/output are the INPUT statement (in two forms) and the PRINT statement (in four forms).

INPUT Statement

The use of the INPUT statement allows data to be entered from the terminal while a program is executing. For example:

```
10 INPUT H,W,L
```

When the line shown above is executed, a question mark (?) will print at the terminal and the system will pause. The data must be entered into the program. The values of H, W, and L are entered from the keyboard. When the carrier return is depressed, the program continues processing, using the entered values of H, W, and L.

MAT INPUT Statement

The MAT INPUT statement allows the user to enter data into matrixes without specifying each element of the matrixes. For example:

```
20 MAT INPUT A
```

When the statement shown above is executed, a question mark will be printed at the terminal, and the system will pause. The data values for the first row of the matrix A must be entered, separated by commas. Then the carrier return must be pressed. Two question marks will be printed at the terminal, requesting values for the next row of A. The user must then enter values for the second row of A, separated by commas and terminated by a carrier return.

Printing of question marks and entry of a value by the user are repeated until values have been entered for all rows of A.

PRINT Statement

The PRINT statement provides output from the program to the terminal. For example:

```
20 PRINT 'THE VALUES OF H, W, AND L ARE'
30 PRINT H, W, L
```

Line 20 would cause the information enclosed within the quotes to be printed at the terminal. Then line 30 would cause the current values of H, W, and L to be printed.

PRINT USING Statement

The PRINT USING statement specifies data output using an IMAGE statement. This allows control of the format of the printed line. For example:

```
 20 PRINT USING 100,H
  .
  .
  .
100 :VALUE OF H IS ###
```

This statement causes the current value of H to be printed, in the format specified by the IMAGE statement identified by line number 100. (For further explanation, refer to the PRINT USING and IMAGE statements in the Reference Information part of this manual.

MAT PRINT Statement

The MAT PRINT statement prints the values of a matrix without the need to specify each element of the matrix.

Consider a three-by-three matrix A with values as follow:

```
A(1,1) = 11        A(1,2) = 12        A(1,3) = 13

A(2,1) = 21        A(2,2) = 22        A(2,3) = 23

A(3,1) = 31        A(3,2) = 32        A(3,3) = 33
```

MAT PRINT USING Statement

The MAT PRINT statement could be used to print the matrix with the following statement:

```
50 MAT PRINT A
```

this would cause the values stored in A to be printed out in row and column order:

```
11                 12                 13

21                 22                 23

31                 32                 33
```

The MAT PRINT USING statement specifies printed output of a matrix using an IMAGE statement. The values of the matrix are printed by row under format control of the image statement. For example:

```
 20 MAT PRINT USING 100,A
  .
  .
  .
100 :VALUES OF A ARE ## ## ##
```

This statement causes the current values of all members of the matrix A to be printed by row, in the format specified by the IMAGE statement identified by line number 100. (See the "MAT PRINT USING Statement" and "IMAGE Statement" in "Part 3. Reference Information" of this manual for details.)

DISK INPUT/OUTPUT

A collection of related data items, treated as a unit, is called a file. There are two types of disk data files in VM/370 BASIC: data files and program data files.

Data Files

Data files are files created during execution of BASIC programs. These files are assigned a filetype of BASDATA by the processor. Data files have undefined length records of up to 3440 characters.

Data files may be created and accessed with the following program statements and terminal commands:

- GET statement
- PUT statement
- OPEN statement
- CLOSE statement
- RESET statement

The GET statement is used to transmit records from the data file to the program. The PUT statement transfers data from the program to the data file. The OPEN statement is used to activate a data file preparatory to data transmission. The OPEN statement associates a data file reference number with a named data file; the named data file is referenced by this number in the GET, PUT, RESET, and CLOSE statements. For example:

```
10 OPEN 21, 'AFILE', INPUT
12 OPEN 22, 'BFILE', OUTPUT
.
.
.
30 GET 21: V, D, T, X, S, F
.
.
.
40 PUT 22: D, T, F
.
.
.
50 CLOSE 21, 22
```

Line 10 opens AFILE as input and assigns it to file number 21. Line 12 opens BFILE as output and assigns it to 22. Line 30 reads six data values as input. The values are stored in the variables V, D, T, X, S, and F, respectively. Line 40 writes output consisting of the three data values from D, T, and F onto file number 22.

A file that has been opened is an active file. A CLOSE statement is used to deactivate the file. Thus, the CLOSE statement in line 50 of the above example causes files 21 and 22 to be deactivated. After a file is closed, it cannot be referenced again until it is reopened.

The opening of a file may be implied in a GET or PUT statement. The example above could have been written:

```
30 GET 'AFILE',V,D,T,X,S,F
 .
 .
 .
40 PUT 'BFILE',D,T,F
 .
 .
 .
50 CLOSE 'AFILE','BFILE'
```

A file can always be referenced by its file name. It can be referenced by a file number only if it has been opened explicitly.

If a file is closed and then reopened, the file pointer is reset to the first record in the file. Normally, a file should be left open until all necessary transmission is completed between the program and the file.

The RESET statement is provided to reset the file pointer to the first record in the file, as required. Four files may be active at any one time (that is, opened but not yet closed).

Program-Data Files

Program-data files are files created using the VM/370 CMS Editor. Record format for program-data files is fixed with a record length of 80. The filetype for program-data files is BASDATA, which must be specified by the user.

The VM/370 terminal commands, line-entry techniques, and library facilities used to create source program files are also used to create, edit, and save program-data files.

File items for program-data files are entered using the following format:

```
┌──────────────────────────────────────────────────────────────────┐
│                                                                  │
│  linenum          constant1[,constant2,...]                      │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

where linenum is the line number of the entry.

constant is a numeric or literal constant being entered in the program-data file.

A program-data file cannot be used for output; that is, it cannot be written or modified by a VM/370 BASIC program.

Although a program-data file may be useful in various situations, execution time and the disk space to store the data may be three times greater than that required for an equivalent data file written by a VM/370 BASIC program.

FILE ALLOCATION

Space allocation for data files is performed dynamically as records are written. The limiting factor is the amount of space available on the user's disk.

A program may specify a data file to be used with a program via the OPEN statement. For example:

```
10 OPEN 4, 'AFILE', INPUT
```

This statement would cause the data file AFILE to be attached to the program as an input file. The file pointer would be set to the first record in the data file. The file could then be accessed by a GET statement in the program.

DATA FILE STORAGE

Data records are stored sequentially in each data file. The storage requirements for data items are:

1. Alphameric data items require 18 bytes.
2. Short-form numeric data items require four bytes.
3. Long-form numeric data items require eight bytes.

Each file can contain a combination of alphameric data items, short-form numeric data items, and long-form numeric data items. Numeric data items are preceded by two bytes which specify data type (short-form or long-form) and number of subsequent data items (255 bytes maximum in a single block). Alphameric data items are preceded by two bytes that specify data type and a length of 18 bytes.

Data records are stored in disk storage units. Many data records may be stored in each storage unit. The maximum amount of storage in each storage unit is 3440 bytes. The maximum number of storage units per file is 3730.

PROGRAM CHAINING

The VM/370 BASIC user has the ability to chain one program to another program in his user disk. This ability is provided by the CHAIN statement, which terminates execution of the current (chaining) program and initiates execution of a specified (chained) program. The CHAIN statement may also be used to specify an argument whose value is passed to the chained program.

The general form of the CHAIN statement is:

```
nnn  CHAIN  program-name[,argument]
```

where *nnn* is the line number.

program-name identifies the chained program.

argument identifies the value to be passed to the chained program. (See CHAIN and USE in the Reference Information part of this manual.)

The length of the value passed to the chained program cannot exceed 16 bytes. It is truncated or blank-filled on the right to 16 bytes when passed to the chained program.

If a VM/370 BASIC program is to be invoked as a chained program and a passed value is to be accessed by it, the chained program must contain a USE statement. The statement may be placed anywhere in the program. Its general form is:

```
nn  USE parameter
```

where *nn* is the line number.

parameter is an alphameric variable. Sixteen bytes of parameter information (plus two rightmost blanks) will be placed in the

designated alphameric variable before the chained program is executed.

The general procedure by which the VM/370 BASIC user initiates program chaining is shown by the following example.

Note that the intermediate READY message, which would normally follow execution of the chaining program, is suppressed.

Total central processor (CPU) time is accumulated and printed in the final READY message.

```
type programa basic

     10 PRINT 'CHAIN FROM '
     20 A$ = 'PROGRAMA'
     30 CHAIN 'PROGRAMB', A$
     40 END

R; T=0.09/0.23 16:54:12

type programb basic

     10 PRINT C$; ' TO PROGRAMB'
     20 USE C$
     30 END

R; T=0.06/0.20 16:54:33

basic programa

CHAIN FROM
PROGRAMA TO PROGRAMB
R; T=0.46/1.14 16:54:49
```

PART 3: REFERENCE INFORMATION

This part of the manual is designed as a reference aid and consists of two sections: "Functional Classification of BASIC Statements" and "Summary of BASIC Statements".

The functional classification simply delineates all the statements of the language, classifying them in functional order and giving a brief definition of each statement.

The summary of BASIC statements lists the statements of the language (including functions and statements specified using operators) and provides the syntax and rules for coding and using them in programs.

FUNCTIONAL CLASSIFICATION OF BASIC STATEMENTS

The charts below functionally group the statements that comprise the BASIC language. A brief definition of the statement usage is also included.

Assignment Statements

LET	Assigns values to variables.
DEF	Defines user-function formats.

Internal Specification Statements

READ	Reads variables into a data table where they are associated with values defined in the DATA statement.
DATA	Constructs a data table containing values to be associated with READ variables.
RESTORE	Resets the data table pointer to the first item in the table.

Terminal I/O Statements

INPUT	Allows programmer to enter data into a program interactively.
PRINT	Prints specified print fields at the user's terminal.
PRINT USING	Prints a formatted print line defined in the user's program.
IMAGE	Allows user to edit dynamically computed values into print lines.

Array Declaration Statements

DIM	Allocates storage for named arrays.

Disk I/O Statements

OPEN	Opens a data file for input or output.
GET	Accesses a data file and associates the file data with variables.
PUT	Outputs data to a specified data file.
CLOSE	Closes a data file for input or output.
RESET	Resets a data file to the beginning of the file.

Program Chaining Statements

CHAIN	Chains one program to another program.
USE	Enables a variable to be passed to a chained program.

Loop Statements

FOR	Defines a loop in a program.
NEXT	Defines the range of a loop.

Branch Statements

GOTO	Directs execution to another place in a program.
IF	Defines a conditional statement.
GOSUB	Directs execution to a subroutine.
RETURN	Directs execution from a subroutine to a calling program.

Matrix Statements

Matrix Addition	Causes elements of matrixes to be added.
Matrix Assignment	Causes values in one matrix to be assigned to corresponding elements in another matrix.
Matrix CON Function	Sets all elements of a matrix equal to 1.
Matrix GET	Reads values into a specified matrix.
Matrix IDN Function	Defines an identity matrix.
Matrix INPUT	Allows data to be read into a matrix during program execution.
Matrix Inversion	Inverts the values in a matrix.
Matrix Multiplication	Causes elements of one matrix to be multiplied by elements in another matrix.
Matrix Multiplication (Scalar)	Causes elements in a matrix to be multiplied by a constant.
Matrix PRINT	Prints the elements in a matrix.
Matrix PRINT USING	Prints the elements in a matrix in a formatted print line.
Matrix PUT	Writes matrix data onto an output file.
Matrix READ	Reads data into a specified matrix.
Matrix Subtraction	Subtracts elements of one matrix from another.
Matrix Transposition	Transposes elements of one matrix onto another.
Matrix ZER Function	Sets all elements of a matrix to 0.

Remarks Statement

REM	Used to document programs.

Pause and Termination Statements

PAUSE	Causes program execution to pause.
STOP	Causes program execution to terminate.
END	Causes program compilation to terminate.

SUMMARY OF BASIC STATEMENTS

This section is provided for quick reference of the statements that comprise the BASIC language. The statements are alphabetically listed. Each description gives a definition of the statement, the syntax for coding it, and rules for using the statement in programs.

CHAIN STATEMENT

The CHAIN statement is used to link an executing BASIC program with another BASIC program existing on disk. The format for coding CHAIN is as follows:

```
CHAIN          pname[,arg]
```

where pname (that is, the VM/370 BASIC filename) is the name of the program being invoked; pname can be either an alphameric variable or a literal constant.

arg is an alphameric variable or a literal constant whose value may be passed to the invoked program.

The CHAIN statement terminates execution of the current program and initiates execution of the chained program.

The program being invoked must be named on the user's disk and must have a filetype of BASIC.

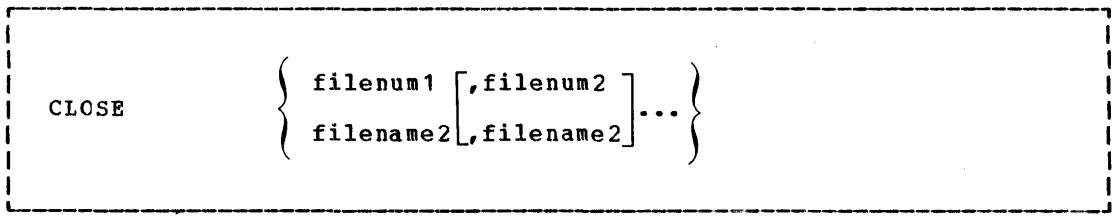
If arg is being passed to the chained program, then that program must contain a USE statement. Also, when arg is specified, it is truncated or blank-filled on the right. It is always 16 characters in length.

The examples below show how to use the CHAIN statement:

```
110  CHAIN  'P1','DATEFILE'
300  CHAIN  '**PROG'
210  CHAIN   A$,B$
```

CLOSE STATEMENT

The CLOSE statement causes a file that is active (open) in a program to be deactivated. It takes the form:



where filenum is an expression that specifies the numbers of the files being closed.

filename that specifies the name of the file to be closed is a literal constant.

The CLOSE statement causes the data file or the program-data file specified by filenum or filename to be removed from the list of active files in a program.

If the specified file is not active, the CLOSE statement is ignored.

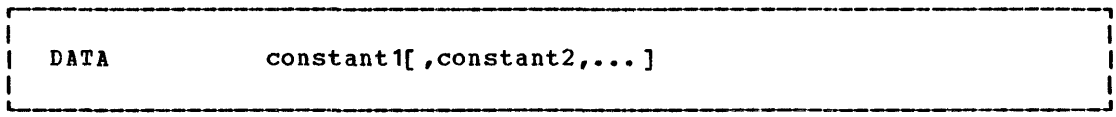
When a program is terminated, all currently active data and program-data files are automatically closed.

The following example shows how to enter the CLOSE statement:

```
100  CLOSE  1,A,200,'FILE'
```

DATA STATEMENT

The DATA statement is used to define values for use in a program. The statement takes the form:



where constant is a numeric or literal constant that defines the value to be used in the program.

During compilation, the BASIC compiler creates a table of the values found in the DATA statement. The values are stacked in the table in order of appearance.

Entries in the DATA table are 18 characters in length. If the entry is larger than 18 characters, it is truncated on the right. If it is smaller than 18 characters, it is padded on the right with blanks. If a literal constant of no characters is entered, it is entered in the DATA table as 18 blank characters.

Generally, the DATA statement is used to define a tables of values which are positionally associated with variables defined in the program by means of the READ statement.

The following examples show how to enter DATA statements:

```
10  DATA   10, 15, 17
20  DATA   34E-51, 532, 3.021, 1E6
30  DATA   'JOHNSON','SMITH','BROWN','JONES'
```

DEF STATEMENT

The DEF statement is used to define user functions. It takes the form:

```
DEF          FNvar1(var2)=ae
```

where var1 must be a simple numeric variable specified by a single letter (or a character from the extended alphabet).

var2 is a simple numeric variable specified by a single letter (or a character from the extended alphabet).

ae is an arithmetic expression.

The function is evaluated by substituting a user expression for each occurrence of the dummy variable var2 into the expression ae, and then evaluating ae.

A function may be defined anywhere in the program (before or after its use).

Other functions may be invoked in DEF statements if no direct or indirect recursive actions are involved. That is, the function being defined (say FNA) may have another function (say FNE) as part of the defining expression, and ae, provided that FNB does not, in turn, have FNA as part of its definition. The following statements show how function maybe defined and evaluated.

```
70 DEF FNB(X)=5*X**2+27
80 DEF FNA(X)=FNB(X)+X**3
.
.
.
140 LET R=FNA(Z)+23
```

Line 140 is equivalent to R=(((5*Z**2+27)+Z**3)+23

DIM STATEMENT

The DIM statement provides the ability to allccate storage space for named arrays and matrixes. It takes the following form:

```
DIM          array1(row1[,col1])[,array2(row2[,col2])...]
```

where arrayi specifies the name of the array for which storage is being allocated.

rowi specifies the number of rows in arrayi.

coli specifies the number of columns in arrayi.

rowi and coli must be unsigned integers.

DIM allocates storage space for named arrays and matrixes and their specified dimensions.

Once an array or matrix has been declared, either implicitly or explicitly, in a program, that array or matrix name may not be used in a DIM statement.

The following is an example of the DIM statement:

```
10 DIM A(10), B(2,3), C(10,50)
```

END STATEMENT

The END statement is used to specify the end of a program. It takes the form:

```
END          [comment]
```

where comment may be any note the programmer wishes to include; it is ignored by the language processor.

END causes program compilation and execution to terminate. Lines following an end statement are ignored by the processor.

When the END statement is omitted the processor supplies the effect of the END statement. An example of the END statement is:

```
100     END
```

FOR STATEMENT

The FOR statement is used in conjunction with the NEXT statement to define loops in BASIC programs. It takes the form:

```
FOR          var=initval TO limit[ STEP increment]
             .
             .
             .
NEXT         var
```

where var defines the index loop and is a simple numeric variable.

initval is an arithmetic expression that specifies the value of var the first time through the loop.

limit is an arithmetic expression that specifies the maximum number of times the loop can be executed.

increment is an arithmetic expression that specifies the value added to the initial value each time the loop is executed.

The range of the loop is defined by the NEXT statement. All statements between the FOR and NEXT statements will be excuted sequentially each time through the loop until the limit specified by limit is reached.

The statements of the loop are executed repeatedly with var equal to initval, then with var equal to initval + increment, and so on, until the value defined by limit is reached. If STEP increment is omitted, an increment value of one is assumed. When the value of increment is zero, the only means of exiting the loop is via the NEXT statement.

The simple numeric variable var is the index of the FOR loop. Throughout the range of the loop, the value of var is available for computation, either as an ordinary variable or as the variable in a subscript, the index is also available for computation when the loop is exited and is equal to the last value it attained.

Branching into the range of the loop is permissable, but should be done carefully.

The following example shows how the FOR loop is used.

```
20  FOR X = 3*Z+6 TO 25 STEP 2/A
 .
 .
 .
60  NEXT X
```

GET STATEMENT

The GET statement is used to retrieve data from a data or program-data file and to assign variable names to the data. The statement takes the following form:

```
GET  { filenum: }
     { filename,} var1[,var2...]
```

where filenum is an arithmetic expression representing the number of the file being accessed.

filename is a literal constant specifying the name of the file being accessed.

var specifies the variable names to be associated with data retrieved from the file being accessed.

File specification in a GET statement may take one of three forms:

- File Name Specification. If the file specification is by file name (filename, above), then the file is opened, if necessary, before the GET is executed.
- File Number. If the file specification is an arithmetic expression (filenum, above), then the truncated integer value of the expression ($1 \leq filenum \leq 255$) is used as the file number. The file number must refer to a file that has been opened by means of an OPEN statement containing that number, and the specified file must be active for input.
- Implied. If no file specification is present, a file number of 1 is assumed. As in the case of the file name specification, the file must be opened for input, otherwise the program is terminated.

The variables, vari, are associated with the input data positionally, that is, var1 will be associated with the first input item from the file, var2 with second input item, etc., etc.

The association process ensures that all the variables specified have an associated value. In the case where there are not as many input items as there are vari, the program is terminated.

If a GET statement specifies a file which is not designated as an input file, the program is terminated.

In the case where vari is a numeric variable, the corresponding data input item must also be numeric; likewise, where the vari is an alphameric variable, its corresponding data item must be a literal constant.

The examples below show how the GET statement is coded.

```
100  GET   'FILE',I,A(I)
100  GET   A,B,C
100  GET   F1:D(I,J)
100  GET   200:X,Y,Z
```

GOSUB STATEMENT

The GOSUB statement provides the ability to transfer control to a subroutine. It takes the form:

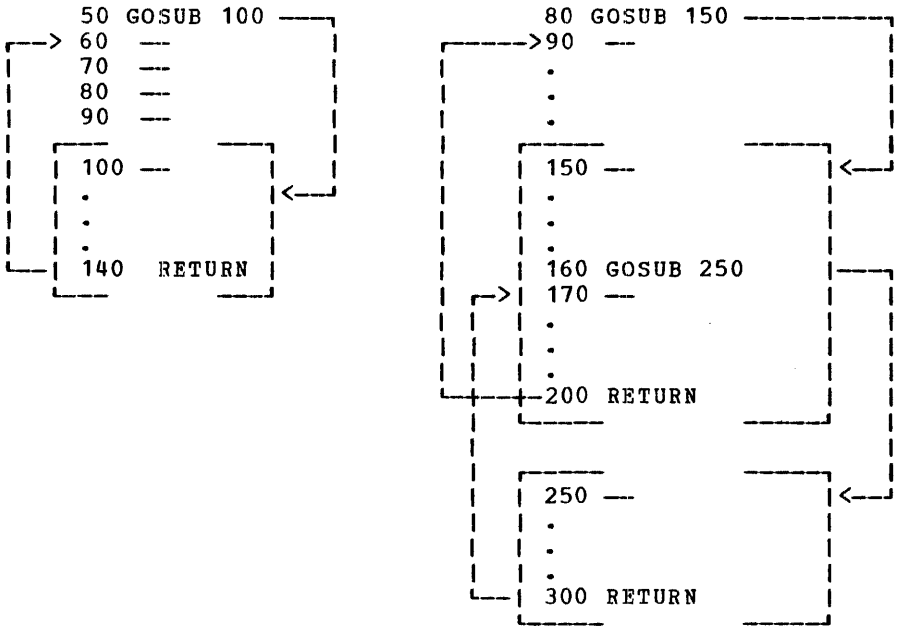
```
┌──────────────────────────────────────────────────────────────────┐
│                                                                  │
│  GOSUB          subname                                          │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

where *subname* is the name of the subroutine to which control is being transferred.

The GOSUB statement transfers control to the subroutine specified by subname.

Control is returned to the calling program by means of the RETURN statement located in the subroutine. When control is returned to the statement following the GOSUB.

The following diagrams show the flow of control through routines and subroutines.



GOTO STATEMENT

The GOTO statement provides the ability to transfer control to a different part of a program.

There are two types of GOTO's in BASIC, the simple GOTO and the computed GOTO.

THE SIMPLE GOTO

The simple GOTO statement transfers control to the specified line in the program. This statement takes the form:

```
GOTO          linenum
```

where linenum is an arithmetic expression specifiying the target line of the GOTO.

Examples of a Simple GOTO's are:

```
30  GOTO   845
80  GOTO    29
```

THE COMPUTED GOTO

The computed GOTO transfers control to a given line depending on the value of an arithmetic expression. It takes the following form:

```
GOTO          linenum1[,linenum2,...]   ON ae
```

where linenumi are line number specifications.

ae is an arithmetic expression.

Control is passed to linenum1 when the truncated integer value of ae is evaluated as one. Control is passed to linenum2 when the truncated integer value of ae is evaluated as two, etc., etc.

When the truncated integer value of ae is evaluated as less than 1 or greater than the highest number of any line in the program, control passes to the next sequential statement after the GOTO.

The following example shows how the computed GOTO works.

```
40  GOTO  34, 60, 1, 34, 10, 45 ON 3-4/X-Z
```

In this example, when 3-4/X-Z is evaluated as either 1 or 4, control will be passed to statement 34; when the expression is evaluated as 2, control passes to statement 60, etc., etc.

IF STATEMENT

The IF statement provides the ability to branch to another statement in a program depending on the evaluation of a relational expression. The statement takes the form:

```
IF        parm1 operator parm2 { GOTO } linenum
                               { THEN }
```

where parmi may be arithmetic expressions, alphameric variables, or literal constants. These parameters are evaluated in the relational expression defined by operator.

operator may be any of the relational operators allowed in BASIC:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
<>	not equal to

linenum is a decimal number specifying the target line to which control is being directed.

parm1 and parm2 must be the same type of BASIC symbol, that is, if parm1 is a numeric variable, parm2 must be a numeric variable also.

When the relational expression specified in the IF statement is evaluated as true, then control is directed to the line number specified in the IF statement. If the expression is false, control drops through the IF statement to the next statement in the program.

Literal constants containing less than 18 characters are padded on the right with blanks; literal constants containing more than 18 characters are truncated on the right. Both padding with blanks and truncation occur before the constants are compared in the expression.

A literal constant with no defined characters is interpreted as 18 blank characters.

Differences in the floating-point approximations of decimal numbers arrived at through different logical steps can affect comparisons. This can be avoided by comparing numbers only within the appropriate range of precision. For example, rather than using the following comparison:

```
IF A<>B THEN 100
```

use of the following statement would be preferable:

```
IF ABS(A-B)=>.001 THEN 100
```

THEN and GOTO have the same effect of directing execution to the target statement. Their use depends only on the programmer's preference.

The following examples show how to code the IF statement:

```
30  IF A(30)>X+2/Z THEN 85
40  IF S1<=37.22 GOTO 67
50  IF A+B>C THEN 80
60  IF R$=A$ GOTO 120
```

IMAGE STATEMENT

The IMAGE statement is used to provide a formatted print line for use with the PRINT USING statement. The IMAGE statement can contain image areas designed to contain dynamically computed data. The statement takes the form:

```
:[charstring1,imagearea1,charstring2,imagearea2,...]
```

where charstringi is a character string which may contain any character except a commercial pound sign, #. It may also be null.

imageareai is the area into which dynamically computed data from the program may be substituted. This area is specified using commercial pound sign, #.

In VM/370, the pound sign (#) is the line-end character and, therefore, IMAGE statements must hide pound signs from the system. The CMS SET command can be used to redefine the line-end symbol. Also, the pound sign can be hidden by means of the CMS Escape Character, the double quote.

In the PRINT USING statement, the user can specify an IMAGE statement which has been formatted for his specific use. In this statement he can specify any number of symbols (variables constants) and associate these positionally with image areas in an IMAGE statement. When the IMAGE statement is printed it will contain the current value of the variables specified in the PRINT USING statement.

In the example below, for instance, the variables A and B represent a question number and the answer to the question, which is a number.

```
20  PRINT USING 30 A,B
30  :THE ANSWER TO QUESTION # is ###.##
```

The colon begins the IMAGE statement; any blanks following the colon and the first character will be included in the printed output.

There are rules for specifying the three types of numbers:

1. I format consisting of an optional sign followed by one or more # characters. For example, ## is an I format specification.

2. F format consisting of an optional sign followed by the optional occurrence of one or more # characters, a decimal point, and the optional occurrence of one or more # characters. There must be at least one # character in the specification. For example, +#.##### is an F format specification.

3. E format is an I format or F format conversion specification followed by either four !!!! characters of four |||| characters

A format consisting only of # characters can be used to print a character variable or literal constant. The character string will be placed into the format left-justified and truncated or blank filled on the right to the length of the format.

The IMAGE statement may also be used with the MAT PRINT USING statement to specify image areas and to format output. See examples of this usage in the section entitled MAT PRINT USING.

INPUT STATEMENT

The INPUT statement provides the ability to associate values with variable names interactively at the terminal. The statement takes the form:

```
INPUT          var1[,var2,...]
```

where *vari* are variable names typed at the terminal.

When the INPUT statement is encountered by the language processor, a question mark is printed at the terminal. Data in the form of numeric and/or literal constants, separated by commas, may then be entered from the terminal.

Where the input line is not long enough to accept all of the data required, that line may be continued by typing a comma and hitting the carrier return.

The specified variables assume the values of the data in order of entry; the number of items entered must equal the number of variables in the INPUT statement list. Numeric constants must be entered for numeric variables; literal constants must be entered for alphameric variables.

If a literal data entry is not empty or does not contain a comma, the entry need not be bounded by quotation marks; leading blanks are ignored but embedded blanks are significant.

A literal constant containing fewer than 18 characters is padded with blanks on the right. A literal constant containing more than 18 characters is truncated on the right. A literal constant containing no characters is interpreted as 18 blank characters.

The following examples show how to use the INPUT statement.

```
10  INPUT X,Y(X),Z(R+3),C1
.
.
.
90  END
basic pgname
? 20,15.5,4,.35
R;
```

The input response shown above causes a value of 20 to be assigned to the variable X, a value of 15.5 to be assigned to Y(20), a value of 4 to be assigned to the element of array Z identified by the computed value of the subscript R+3, and a value of .35 to be assigned to C1.

```
10  INPUT A$,R
.
.
.
90  END
basic pgname
? YES,20
R;
```

The input response shown above causes the literal constant YES to be assigned to the alphameric variable A$, and the numeric constant 20 to be assigned to the numeric variable R.

LET STATEMENT

The LET statement provides the ability to assign a value to a variable or a series of variables. The statement takes the form:

```
[LET]          var1,[,var2,...]=value
```

where *vari* are variables.

value may be an arithmetic expression, and alphameric variable, or a literal constant.

The variables on the left of the equal sign assume the value specified on the right of the equal sign.

If value is an arithmetic expression, all vari must be numeric.

If value is an alphameric variable or a literal constant, then vari must be alphameric.

All assignment of value and subscript evaluation take place serially, from left to right. Thus, I,A(I)=7 is equivalent to I=7 and A(7)=7.

The actual specification of LET is optional, as shown below in lines 30 and 60. Lines 50 and 60 below are equivalent.

```
20  LET A1 = Z(3)/Y(A+4)
30  X1=49+Z(4)
40  LET A=5
50  LET G$ = 'MONDAY'
60  G$ = 'MONDAY'

110 LET Y, X(Y+3), Z, X(42) = 10.0967
120 LET A, B, C, D, E, F, = 0.0
130 LET D$, T$, P$, = B$
```

The simple numeric variables Y AND Z, and the thirteenth and forty-second elements of the numeric array X are assigned the value 10.0967. The numeric variables A, B, C, D, E, and F are set to 0.0; and the alphameric variables D$, T$, and P$ are set to the current value of the alphameric variable B$.

MATRIX ADDITION FORMAT

BASIC provides for the addition of corresponding elements in matrixes. The sums resulting from these additions are then placed in a specified matrix. The format for coding this is:

```
MAT          matrix1 = matrix2 + matrix3
```

where *matrixi* are matrixes of identical dimensions.

If the matrixes are not conformable (having dimensions consistent with the rules of matrix algebra), program execution is terminated.

The example below shows how matrixes can be added.

```
20  MAT  A = A+B
```

MATRIX ASSIGNMENT FORMAT

BASIC provides the capability to equate all the elements in a matrix to corresponding elements in another matrix. The statement takes the form:

```
MAT          matrix1 = matrix2
```

where *matrixi* must be matrixes of identical dimensions.

This statement assigns the values of the elements of matrix2 to the corresponding elements of matrix1.

The current dimensions of matrix1 and matrix2 must be identical. Otherwise, the program is terminated.

An example of the matrix assignment statement is:

```
20  MAT  A = B
```

MATRIX CON FUNCTION

The matrix CON function provides the ability to set all the elements of a specified matrix to the value 1. Also, the CON function may be used to redimension the matrix. The form for using the CON function is:

```
MAT          matrix = CON   [(dim1[,dim2])]
```

where *matrix* specifies the target matrix for the function.

dimi specify new dimensions for the matrix.

When CON is encountered, all the elements of matrix are redefined to the value 1.

For information on how dimi are used to redimension a matrix, see the section entitled "Matrix Manipulation".

The examples below show how the matrix CON function is used.

```
20  MAT  A = CON
30  MAT  B = CON(J,K)
```

MAT GET STATEMENT

The MAT GET statement allows data to be read into a specified matrix (or matrixes) without referencing each member of the matrixes individually. The statement takes the form:

```
MAT GET [filenum:  ]matrix1[ (dim1[ ,dim2 ]) ][ ,matrix2[ (dim1[ ,dim2 ]) ]]...
        [filename, ]
```

where filenum is an expression specifying the value of the file number being invoked.

filename is the name of the file being invoked.

matrixi specifies the matrix (or matrixes) whose elements are being defined.

dimi are dimension specifications used to redimension the respective matrixes.

The MAT GET statement is similar to the GET statement. It allows numeric data from an INPUT file to be read into matrixes and associated with the matrix elements without referencing each element separately.

Elements are read by rows from the specified data file or program-data file until the matrix is filled. If the data file is exhausted before the matrix if filled, the program is terminated.

If the data file or program-data file specified by the MAT GET is not active or not an output file, program execution is terminated.

Note that the filenum/filename specification is optional; if the specification is omitted, its value defaults to 1. If the truncated integer value of filenum is less than 1 or greater than 255, program execution is terminated.

For information on matrix redimensioning, see the section "Matrix Manipulation".

The following statements are examples of the MAT GET statement.

```
20  MAT GET Z(E-3): A,B,C(10,K)
30  MAT GET A
40  MAT GET 'FILE',C,B
```

MATRIX IDN FUNCTION

The matrix IDN function is used to redefine the form (that is, the dimensions) of a matrix by means of an identity matrix. The function takes the form:

```
MAT          matrix = IDN[(dim1[,dim2])]
```

where matrix is the name of the matrix whose form is being redefined.

dimi specify the dimensions which may be used to redimension matrix.

If matrix is not conformable, program execution is terminated.

The following statements are examples of the IDN function.

```
20  MAT A = IDN
30  MAT B = IDN(4,4)
```

See the section "Matrix Manipulation" for information on matrix redimensioning.

MAT INPUT STATEMENT

The MAT INPUT statement provides the ability to enter numeric values from the terminal into specified matrixes without referencing each member of each matrix separately. The statement is similar to the INPUT statement and takes the form:

```
MAT INPUT       matrix1[(dim1[,dim2])][,matrix2[(dim1[,dim2])]...
```

where matrixi specify the matrix or matrixes into which data is be entered.

dimi specify the dimension(s) used to redimension a given matrix.

When the MAT INPUT is encountered by the language processor, a question mark is printed at the terminal. At that point the programmer types in values to be inserted into the matrix.

Values are entered by row. If an input line is not large enough to contain a row, the line may be continued by typing a comma as the last character on the line and hitting the carrier return.

When the row has been completed, the system requests input for the second row by printing two question marks. The user enters input values for the row.

Printing of two question marks for each new row and single question marks for the completion of a row, interspersed with user entry of input values, continues until all data has been entered.

The number of values entered for a row must equal the number of elements in a row of the matrix.

Only numeric entries are valid.

See "Matrix Manipulation" for an explanation of redimensioning.

```
type test basic
10  DIM A(10,10)
20  MAT INPUT A(2,2)
 .
 .
 .
90  END

basic test

?  1,2

?? 3,4
```

The input shown above causes the numeric constants 1 and 2 to be taken as the values of the elements in the first row of the matrix A. The numeric constants 3 and 4 are taken as the values of the elements in the second row of A.

MATRIX INVERSION FUNCTION

The matrix inversion function causes one matrix to be replaced by the inverse of another matrix. It takes the form:

```
MAT          matrix1 = INV(matrix2)
```

where *matrixi* are matrixes.

If the matrixes are not conformable, program execution is terminated.

matrix1 cannot be identical to matrix2.

The following statement is an example of the INV function.

```
20  MAT A = INV(C)
```

MATRIX MULTIPLICATION FORMAT

BASIC provides the ability to multiply the elements of a matrix by the corresponding elements in another matrix. The following form is used to multiply matrix elements:

```
MAT          matrix1 = matrix2 * matrix3
```

where matrixi are matrixes.

This statement causes the elements of matrix1 to be replaced by the results of the multiplication of the corresponding elements in matrix2 and matrix3.

The following statement is an example of matrix multiplication.

```
20  MAT  Q = P * R
```

MATRIX MULTIPLICATION (SCALAR) FORMAT

Scalar multiplication of a matrix is permitted in BASIC. That is, each element in a matrix is multiplied by a value and the results are placed in another matrix. The statement takes the form:

```
MAT          matrix1 = (expression) * matrix2
```

where matrixi are matrices.

expression is an arithmetic expression whose value is multiplied times each element of the matrix.

This statement causes each element of matrix1 to be replaced by each corresponding element of matrix2 multiplied by expression.

expression is evaluated before any scalar multiplication.

If the matrixes are not conformable, program execution is terminated.

The example below shows how to use scalar matrix multiplication.

```
20  MAT  A = (A(3,2))*D
```

MAT PRINT STATEMENT

The MAT PRINT statement is used to print the contents of a matrix after the contents have been converted to a specified output format. The format for the statement is:

```
MAT PRINT     matrix1termchar1[matrix2termchar2...[termcharN]]
```

where matrix are matrixes.

termchari are terminal characters; commas or semicolons.

This statement causes each element of each specified matrix to be converted to a specified output format and then printed. After the element has been printed, the carrier is positioned as specified by the terminator character.

Rules for printing as described under the PRINT statement apply for the MAT PRINT statement; however, literal strings are not allowed, and an omitted final terminator character is treated as a comma.

The matrix is printed in order by rows. All of the elements of a row are printed on as many print lines as are required with single line spacing.

A blank print line is used to separate rows.

Printing of the first element of a row always starts at the beginning of a new print line.

The following statement is an example of the MAT PRINT statement.

```
20  MAT PRINT A,B,C
```

MAT PRINT USING STATEMENT

The MAT PRINT USING statement causes each element of specified matrixes to be edited into print lines as directed by IMAGE statements. The statement takes the format:

```
MAT PRINT USING     linenum,matrix1[matrix2,...]
```

where linenum is the line number of the target IMAGE statement.

matrixi specify matrixes.

The first row of a matrix is separated from the line immediately preceding it by one blank line. A blank print line separates succeeding rows of the matrix. Each element of the matrix is edited row by row into the corresponding conversion specification in the IMAGE statement.

The first element of a row is printed at the beginning of a new print line according to the first entry of the corresponding IMAGE statement. The IMAGE statement is reused to complete a row on the next line if necessary.

The following example shows how MAT PRINT USING can be used.

```
 10  DIM A(2,3)
 20  DATA  1,10,33,2,20,44
 30  MAT READ A
 .
 .
 .
100  PRINT USING 300
110  MAT PRINT USING 400,A
300  :                        X          Y
400  : TEST CASE ##    ##.###     ##.####!!!!
```

The printout resulting from this sequence of statements appears as shown by the representative printout below.

```
                        X          Y

TEST CASE  1     10.000     33.0000E+00

TEST CASE  2     20.000     44.0000E+00
```

MAT PUT STATEMENT

The MAT PUT statement causes specified matrixes to be written on an output data file without referencing each element of the matrix individually. It takes the form:

```
MAT PUT [filenum: ]matrix1[,matrix2,...]
        [filename,]
```

where filenum is an expression whose value specifies the file number of the target file.

filename is a literal constant which names the target file.

matrixi specify the matrix(es) from which data is to be read.

The elements from the specified matrixes are written in row order to the output file.

If the data file is not active or has not been specified as an output data file, program execution is terminated.

If the data file is not large enough to contain the matrix elements, program execution is terminated.

The following statements are examples of the MAT PUT statement.

```
20  MAT PUT 2: X, Y, Z
30  MAT PUT A,B
40  MAT PUT 'FILE',A,B
```

MAT READ STATEMENT

The MAT READ statement allows numeric data to be read into the specified matrixes without referencing each member individually. The statement takes the form:

```
MAT READ     matrix1[ (dim1[ ,dim2 ]) ][ ,matrix2[ (dim1[ ,dim2 ]) ]... ]
```

where *matrixi* specifies a matrix or matrixes into which data is to be read.

dimi are expressions used to redimension the matrixes. (See "Matrix Manipulation" for a discussion of matrix redimensioning.)

Elements are read in row order from data tables created by DATA statements. If the data table is exhausted before a specified matrix is filled, program execution is terminated.

The statement below is an example of the MAT READ statement.

```
20  MAT READ  A(J,K),B,C(H)
```

MATRIX SUBTRACTION FORMAT

The BASIC language processor permits matrix subtraction. The subtraction is performed by subtracting elements of one matrix from the corresponding elements of another and placing the results in a specified matrix. The statement takes the form:

```
MAT          matrix1 = matrix2 - matrix3
```

where matrix1 is a matrix used to contain the results of the subtraction.

matrix2 is a matrix from which corresponding elements in matrix3 are subtracted.

matrix3 is a matrix, the values of whose elements are subtracted from matrix2.

If the matrixes are nonconformable, program execution is terminated.

The example below shows how matrix subtraction is used.

```
20  MAT  D = A - B
```

MATRIX TRANSPOSITION FUNCTION

The matrix transposition function causes the elements of one matrix to be transposed onto the elements of another. It takes the form:

```
MAT          matrix1 = TRN(matrix2)
```

where matrix1 is a matrix whose values are replaced by the values of matrix2.

matrix2 is a matrix which is transposed onto matrix1.

matrix1 may not be the same as matrix2.

The example below shows how the TRN function is used.

```
20  MAT D = TRN(X)
```

MATRIX ZER FUNCTION

The matrix ZER function causes all the elements of the matrix specified to assume the value zero. It takes the form:

```
MAT          matrix = ZER[(dim1[,dim2])]
```

where matrix specifies the matrix whose values are being zeroed.

dimi are expressions used in redimensioning the matrix. (See "Matrix Manipulation" for a discussion of redimensioning.)

The examples below show how the ZER function is used.

```
30  MAT A = ZER
40  MAT B = ZER(A+D,Q**Z)
```

NEXT STATEMENT

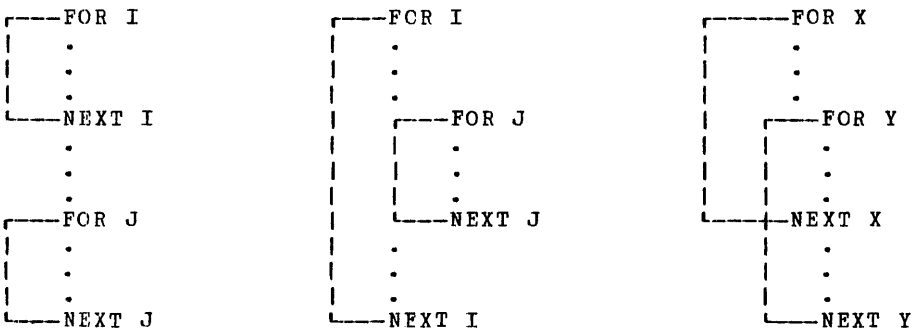
The NEXT statement is used in conjunction with the FOR statement to define the range of a program loop. It takes the form:

```
NEXT          var
```

where var is a simple numeric variable identical to the corresponding variable in the matching FOR statement.

The statement terminates the range of a loop.

The examples below show how FOR and NEXT are used to define multiple loops and nested loops; there is also an example of an invalid loop definition.



Multiple Loops Nested Loops Invalid

OPEN STATEMENT

The OPEN statement is used to activate a data file for input or output or a program-data file for input. The statement takes the form:

```
OPEN          filenum,filename {INPUT }
                               {OUTPUT}
```

where filenum is an expression whose value is assigned to the file number for reference purposes.

filename is an alphameric variable or a literal constant which is the name of the file being opened.

The INPUT and OUTPUT options specify whether the file is being opened for input or output.

The statement causes the data file or program-data file specified by filename to be assigned the file number specified by filenum.

The status of the file is set to active and the pointer is reset to the beginning of the file.

Data files can be opened for either input or output. Program-data files can be opened only for input.

If the truncated integer value of filenum is less than 1 or greater than 255, program execution is terminated.

If the OPEN statement references any file that is currently active, that file is closed and reopened.

If an attempt is made to open more than four concurrent files, program execution is terminated. (See CLOSE.)

If any program-data file is to be used by a program, a program-data file must be the first file opened in the program (so that input areas are of sufficient size). Otherwise, program execution is terminated when an attempt is made, later in the program, to open a program-data file.

The following statements are examples of the OPEN statement.

```
100 OPEN F1,'SYSIN',INPUT
100 OPEN 1,A$,OUTPUT
```

PAUSE STATEMENT

The PAUSE statement causes program execution to halt and a line to be printed at the terminal notifying the user of the halt. The statement takes the form:

```
PAUSE          [comment]
```

where comment can be any note the programmer desires; it is ignored by the language processor.

When the language processor encounters the PAUSE statement, the following line is printed out at the user's terminal:

```
PAUSE AT LINE nn
```

where nn is the line number of the PAUSE statement.

Execution resumes when the user presses the carrier return or by entering any character string and then pressing the carrier return.

The following example shows how PAUSE is used.

```
50     PAUSE
```

PRINT STATEMENT

The PRINT statement is used to direct the printing of information at the user's terminal. It takes the form:

```
PRINT          prfield1tchar1[prfield2tchar2...[tcharN]]
```

where prfieldi are print fields used to specify the information to be printed. Print fields may contain expressions, alphameric variables, and literal constants.

tchari are terminal characters, that is, the comma or the colon. The comma indicates that another print field follows; the colon indicates the end of a print line.

PRINT LINES AND PRINT FIELDS

Print lines are lines being typed in at the terminal specifying the type and format of information the programmer wants printed.

Print lines are comprised of print fields, which are variable length fields containing the programmer's specifications for the type of data he wants calculated and printed.

PRINT ZONES

Each print line is divided into print zones of either of two types: full or packed.

A full print zone is always comprised of 18 characters.

The size of a packed print zone depends on the length of the print field it holds, as shown below:

Print Field Length	Packed Zone Length
2–4 characters	6 characters
5–7 characters	9 characters
8–10 characters	12 characters
11–13 characters	15 characters
14–16 characters	18 characters

The first character of a packed print zone is always reserved for a + or –; default is +.

RULES FOR USING PRINT FIELDS AND PRINT ZONES

If the print field is an alphameric variable, the size of a packed print zone is 18 characters minus the number of trailing blanks. If the print field is a literal constant, the size of a packed print zone equals the size of the converted field. The print field is printed at the terminal as described below.

1. If the print field is an alphameric variable or a literal constant and:

   - tchar is a comma with at least 18 spaces remaining on the print line, printing starts at the current carrier position. If the end of the print line is encountered before the print field is exhausted, printing of remaining characters starts on the next print line.

   - tchar is a comma with less than 18 spaces remaining on the print line, printing starts at the beginning of the next line.

   - tchar is a semicolon, printing starts at the current carrier position. If the end of the print line is encountered before the field is exhausted, printing of remaining characters starts on the next line.

2. If the print field is an expression, printing starts at the current carrier position unless the print line does not contain sufficient space to accommodate the value. In such cases, printing starts at the beginning of the next line.

After the converted print field has been printed, the carrier is positioned as specified by the terminator character.

1. When the print field is an expression or an alphameric variable, the carrier is positioned as follows:

   - If the terminator character is a comma, the carrier is moved past any remaining spaces in the full print zone; if the end of the print line is encountered, the carrier is moved to the beginning of the next print line.
   - If the terminator character is a semicolon, the carrier is moved past any remaining spaces in the packed print zone; if the end of the print line is encountered, the carrier is moved to the beginning of the next print line.
   - If the terminator character is omitted, and the print field is the last print field in the statement, the carrier is moved to the beginning of the next print line.

2. When the print field is a literal constant, the carrier is positioned as follows:

   - If the terminator character is a comma, the carrier is moved past any remaining spaces in the full print zone; if the end of the print line is encountered, the carrier is moved to the beginning of the next print line.
   - If the terminator character is a semicolon, the carrier is not moved unless at the end of the print line, in which case the carrier is moved to the beginning of the next print line.
   - If the terminator character is omitted, and the print field is the last print field in the statement, the carrier is moved to the beginning of the next print line.

3. When the print field is null, the carrier is positioned as follows:

   - If the terminator character is a comma, the carrier is moved 18 spaces; if the end of the print line is encountered, the carrier is moved to the beginning of the next print line.
   - If the terminator character is a semicolon, the carrier is moved three spaces; if the end of the print line is encountered, the carrier is moved to the beginning of the next print line.
   - If the terminator character is omitted, the carrier is moved to the beginning of the next line.

The following examples show how PRINT is used.

```
50 PRINT "X= "; 5, -6.78; (X(2)+4*Z)
60 PRINT Y$
```

PRINT USING STATEMENT

The PRINT USING statment is used in conjunction with the IMAGE statement to edit specified print fields into a print line defined by the IMAGE statement. The PRINT USING statement takes the form:

```
PRINT USING          linenum[prfield1,prfield2,...]
```

where linenum is the line number of the IMAGE statement to be used.

prfieldi specify the values to be edited into the corresponding print fields in the IMAGE statement. prfieldi may be alphameric variables, expressions, or literal constants.

The print fields specified (prfieldi) correspond positionally with the print fields defined in the target IMAGE statement.

If the number of print fields in the PRINT USING statement exceeds the number of conversion specifications in the IMAGE statement, a carrier return occurs at the end of the IMAGE statement and the IMAGE statement is reused for the remaining print fields.

If the number of print fields in the PRINT USING statement is less than the number of conversion specifications in the IMAGE statement, the print line is terminated at the first unused conversion specification.

When the carrier is not positioned at the beginning of a new print line, a carrier return occurs before printing of the edited print line. After the edited print line is printed, the carrier is positioned at the beginning of the next print line.

Each print field is converted to output format as follows:

1. The meaning of an alphameric variable or a literal constant is extracted from the specified string and edited into the print line, replacing all of the elements in the conversion specification (including sign, #, decimal point, and ! or |). If the edited string is shorter than the conversion specification, blank padding occurs to the right. If the edited string is longer than the conversion specification, truncation occurs to the right. A null string results in blank padding of the entire conversion specification.

2. An expression is converted in accordance with its conversion specification:

   a. If it contains a plus sign and the expression value is positive, a plus sign is edited into the print line.

   b. If it contains a plus sign and the expression value is negative, a minus sign is edited into the print line.

c. If it contains a minus sign and the expression value is positive, a blank is edited into the print line.

d. If it contains a minus sign and the expression value is negative, a minus sign is edited into the print line.

e. If it does not contain a sign and the expression value is negative, a minus sign is edited into the print line in front of the first printed digit, and the length of the conversion specification is reduced by one.

f. The expression value is converted according to the type of its conversion specification:

   I-format--the value of the expression is converted to an integer, truncating any fraction.

   F-format--the value of the expression is converted to a fixed-point number, rounding the fraction or extending it with zeros in accordance with the conversion specification.

   E-format--the value of the expression is converted to a floating-point number with one decimal digit to the left of the decimal point, rounding the fraction or extending it with zeros in accordance with the conversion specification.

   If the length of the resultant field is less than or equal to the length of the conversion specification, the resultant field is edited, right-justified, into the print line. If the length of the resultant field is greater than the length of the conversion specification, asterisks are edited into the print line instead of the resultant field.

The example below shows how the PRINT USING statement is used.

```
10 A$ = 'LOSS'
20 B  = 42.0399
30 PRINT USING 40,A$,B
40 :RATE OF #### EQUALS ####.## POUNDS.
```

The printout resulting from this sequence of statements is shown below:

```
RATE OF LOSS IS 42.04 POUNDS
```

PUT STATEMENT

The PUT statement is used to direct that specified values be placed on an output data file. The statement takes the form:

```
PUT [filenum: ] data1[,data2,...]
    [filename,]
```

where filenum is a number specifying the output data set.

filename is a literal constant specifying the name of the output data set.

datai may be an expression, an alphameric variable, or a literal constant; datai specify the data items to be written to the output data set.

The file specification may take one of three forms:

1. File name - If the file specification is a file name, enclosed in quotes, followed by a comma, then the file is opened if necessary before the PUT is executed.

2. File number - If the file specification is an expression followed by a colon, then the truncated integer value of the expression ($1 \leq exp \leq 255$) is used as the file number. The file number must refer to a file that has been opened by means of an OPEN statement containing that number, and that file must still be active for output.

3. Implied - If no file specification is present, a file number of 2 is assumed. As in 2 above, the file must be open for output. If not, program execution is terminated.

A literal constant containing less than 18 characters is padded with blanks on the right. A literal constant containing more than 18 characters is truncated on the right. A literal constant containing no characters is interpreted as 18 blank characters.

If a PUT statement is executed when the specified data file is not active or is assigned as an input file, program execution is terminated. If a PUT statement is executed that causes the size of the data file to be exceeded, program execution is terminated.

Note: Use of the filename in PUT statements should be avoided if the program contains an OPEN statement for logical file number 2 as an output file. In this case, it is not clear whether the default file number 2 or the literal constant (filename) is being referenced. (See execution exception message "FILE REFERENCE UNCLEAR".)

The examples below show how the PUT statement is coded.

```
30 PUT F1: Z3, 5*A-7, A, C, W$
40 PUT 2: 'DATA','STAT','LOG1'
50 PUT 1,2,B,C,(1,3)
60 PUT 'FILE',A
```

READ STATEMENT

The READ statement is used in conjunction with the DATA statement. It reads values defined in the DATA table, associating variables with those values. The READ statement takes the form:

```
READ          var1[,var2,...]
```

where vari are variables.

The variables specified are assigned the next n values in the data table, and the data table pointer is updated accordingly.

If a READ statement is executed when insufficient data remains in the data table, program execution is terminated.

Numeric variables must correcpond to numeric data and alphameric variables must correspond to literal data.

The examples below show how the READ statement is coded.

```
10 READ A,B,C
20 READ Z(4),Z(5),A(K)
```

REM STATEMENT

The REM statement is used to add comments to a program. It takes the form:

```
REM           [comments]
```

where comments may be any character string.

If a GOTO, GOSUB, or THEN uses a REM statement as a target, then the next executable statement after the REM will be executed.

RESET STATEMENT

The RESET statement is used to reset a data file to the first element in that file. It takes the form:

```
RESET [filenum1 ][,filenum2 ]...
      [filename1][,filename2]
```

where filenumi specify the file number(s) being reset to the first element.

filenamei specify the names of the files being reset to the first element.

This statement causes the data file or program-data file specified by the value of filenum to be reset to the beginning of the file. A subsequent GET or PUT statement references the first item in the file.

If filenum is specified, but its truncated integer value is less than 1 or greater than 255, program execution is terminated.

If neither a file number nor a file name is specified, the value 1 is assumed.

If a specified data file is not active, the RESET command is ignored.

The example below shows how to use RESET.

```
100 RESET F1,F2,'FILE'
```

RESTORE STATEMENT

The RESTORE statement causes the next READ statement to begin reading at the first DATA element in the program. It takes the form:

```
RESTORE          [comments]
```

where comments may be any character string.

The example below shows how to use RESTORE:

```
50    RESTORE GO BACK TO ONE
```

RETURN STATEMENT

The RETURN statement is used to transfer control out of a subroutine back to the calling program. It takes the form:

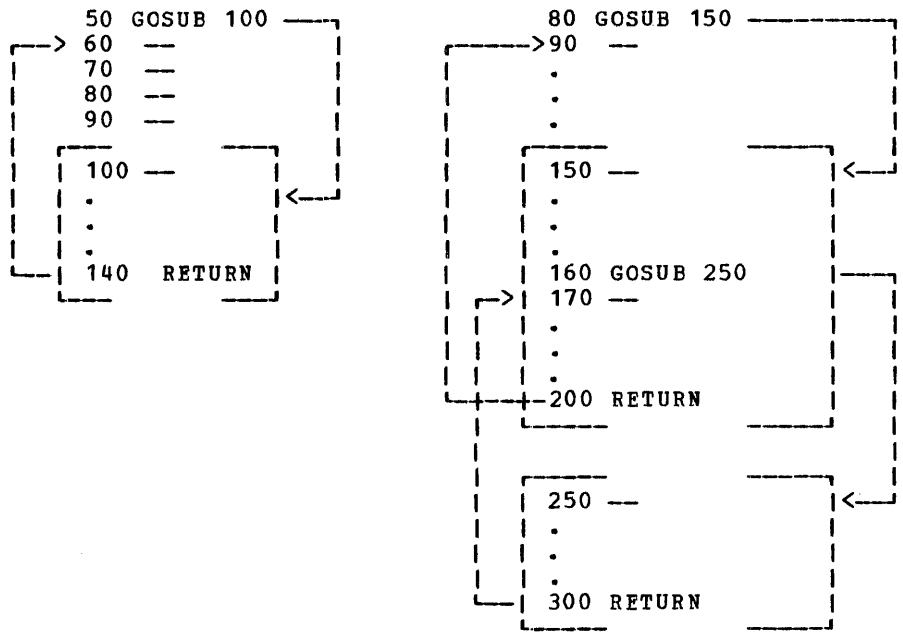
```
RETURN           [comments]
```

where comments can be any character string.

RETURN is the means of exiting from a subroutine. RETURN transfers control to the statement following the last GOSUB executed.

More than one GOSUB statement may be executed before a RETURN statement is executed, but when a RETURN statement is executed there must be at least one active GOSUB. The character string comments may be entered as a comment; it is ignored during compilation and execution.

GOSUB/RETURN examples:



STOP STATEMENT

The STOP statement causes program execution to terminate. It takes the form:

```
STOP          [comments]
```

where comments can be any character string; it is ignored by the processor.

USE STATEMENT

The USE statement is used in conjunction with the CHAIN statement to pass a value to a chained program. It takes the form:

```
USE          var
```

where var is an alphameric variable.

This statement causes the specified alphameric variable to be initialized to the value of the argument named in the CHAIN statement causing this program to be executed. The CHAIN statement itself appears in the chaining program.

The USE statement may be placed anywhere in the chained program.

If more than one USE statement is included in a program, the one having the highest line number will be accepted as valid; others will be ignored.

Sixteen bytes of parameter information (plus two rightmost blanks) will be placed in the designated variable before the chained program is executed.

If execution of the program containing the USE statement was not initiated by means of a CHAIN statement, or if the CHAIN statement contained no argument, the USE statement is ignored. The variable var retains its initial value of 18 blanks.

Example:

```
110 USE A$
```

USER PROGRAM LIMITS

Program Element	Limit
Statement lines	800 [1]
Source characters	64,000 Max [1]
Object program size	114,688 bytes
Array storage (included in program size)	28,668 bytes
Number of image statements	25
Number of FOR loops:	
program limit	80
nest limit	15
Number of function references and GOSUB's per nest	47
Number of files open at once	4
Number of storage units per file	size of user's disk [2] or 3730 storage units
Disk storage unit size	3440 bytes

---

[1] Limit is determined by whichever limit is reached first. The actual limit on source characters is determined by the size of the virtual machine, with a maximum of 64,000 characters. A machine size of 268K allows the full 64,000 character maximum.

[2] An approximation of the remaining storage unit capacity on a disk can be found by dividing the current number of records available (obtained from the statistics generated by the QUERY DISK mode command) by 4.3 (the number of records required to contain one BASIC data storage unit).

INTRINSIC FUNCTION LIMITS

The following table gives the allowable limits for the arguments passed to intrinsic functions.

	Valid Arguments (Minimum < x < Maximum)			
	Short-Form Arithmetic		Long-Form Arithmetic	
Function	Min. Value	Max. Value	Min. Value	Max. Value
SIN(x)	$-PI*2^{18}$	$PI*2^{18}$	$-PI*2^{50}$	$PI*2^{50}$
COS(x)	$-PI*2^{18}$	$PI*2^{18}$	$-PI*2^{50}$	$PI*2^{50}$
TAN(x)	$-PI*2^{18}$	$PI*2^{18}$	$-PI*2^{50}$	$PI*2^{50}$
COT(x)	$-PI*2^{18}$	$PI*2^{18}$	$-PI*2^{50}$	$PI*2^{50}$
SEC(x)	$-PI*2^{18}$	$PI*2^{18}$	$-PI*2^{50}$	$PI*2^{50}$
CSC(x)	$-PI*2^{18}$	$PI*2^{18}$	$-PI*2^{50}$	$PI*2^{50}$
ASN(x)	-1	1	-1	1
ACS(x)	-1	1	-1	1
ATN(x)	-1E75	1E75	-1E75	1E75
HSN(x)	-174.673	174.673	-174.673	174.673
HCS(x)	-174.673	174.673	-174.673	174.673
HTN(x)	-1E75	1E75	-1E75	1E75
DEG(x)	-1E75	1E75	-1E75	1E75
RAD(x)	-1E75	1E75	-1E75	1E75
EXP(x)	-180.218	174.673	-180.218	174.673
ABS(x)	-1E75	1E75	-1E75	1E75
LOG(x)	0	1E75	0	1E75
LTW(x)	0	1E75	0	1E75
LGT(x)	0	1E75	0	1E75
SQR(x)	0	1E75	0	1E75
RND(x)	-1E75	1E75	-1E75	1E75
INT(x)	-1E75	1E75	-1E75	1E75
SGN(x)	-1E75	1E75	-1E75	1E75

APPENDIX B: VM/370 BASIC ERROR MESSAGES

Error messages are printed at the terminal if program syntax and structure errors are detected, program limitations are exceeded, or execution errors or exceptions occur. Many of these conditions are detected at compile time; others cannot be diagnosed until program execution. If a problem persists, save the terminal output, enter CP and request a dump of storage, and contact IBM for programming support.

Error messages are generated from two sources: 1) the VM/370 BASIC Interface, and 2) the CALL-OS BASIC compiler. The interface messages refer to preliminary command syntax checks and virtual machine limitations while the BASIC compiler messages apply to compilation and execution errors.

VM/370 BASIC INTERFACE ERROR MESSAGES

The types of BASIC Interface messages are identified by action codes as follows:

Type	Meaning
W	Warning
E	Error
S	Severe Error
T	Terminal Error

When a condition arises during the execution of a command resulting in a Warning, Error, Severe Error, or Terminal Error message, the command will pass a nonzero return code in register 15. CMS return codes are identified in the following BASIC interface messages as "RC=xx". A description of the assignment of CMS return codes is included in the IBM Virtual Machine Facility/370: System Messages Manual, Order No. GC20-1808.

DMSBSC001E NO FILENAME SPECIFIED

Explanation: The command requires the specification of a filename.

System Action: RC = 24
Execution of the command terminates. The system remains in the same status as before the command was entered.

User Action: Retype the command, specifying the filename.

DMSBSC002E FILE 'fn ft' NOT FOUND

Explanation: The specified file was not found on the accessed disk(s). Either the file does not reside on this disk, the file identification has been misspelled, or incomplete identification has been provided to cause the appropriate disk to be searched. (See the IBM VM/370 Command Language User's Guide, Order No. GC20-1804, for a description of the file identification required by each command and the search procedure used.)

System Action: RC = 28
Execution of the command terminates. The system remains in the same status as before the command was entered.

User Action: To make sure the file exists issue STATE fn ft * or LIST fn ft *. Correct and reenter the command.

DMSBSC003E INVALID OPTION 'option'

Explanation: The specified option appeared illegally in the option list of the command. It may have been misspelled or, if the option is truncatable, it may have been truncated improperly.

System Action: RC = 24
Execution of the command terminates. The system remains in the same status as before the command was entered.

User Action: Check the command line and try again.

DMSBSC007E FILE 'fn ft' IS NOT FIXED, OR GREATER THAN 256 CHAR. RECORDS

Explanation: The input file must have fixed length records of up to 256 characters each in order to execute the command.

System Action: RC = 32
Execution of the command terminates. The system remains in the same status as before the command was entered.

User Action: The record format and length may be corrected using the COPYFILE command.

DMSBSC025E NULL SOURCE LINE. COMPILER TERMINATED

Explanation: There is a blank line in the source program.

System Action: RC = 32
Execution of the command terminates. The system remains in the same status as before the command was entered.

User Action: Edit the BASIC source file and delete the blank line.

DMSBSC104S ERROR READING FILE 'fn ft' FROM DISK

Explanation: An unrecoverable error occurred while reading the file from disk. This error may be caused by any one of the following conditions:

- Given file not found.
- Buffer area not within user storage limits.
- Permanent disk read error.
- Number of records ≤ 0 or > 32768.
- Fixed/variable flag in file status table entry is not F or V.
- Given memory area was smaller than actual size of the records read. (This error is legitimate if reading the first portion of a large record into a little buffer. It does not cause the function to terminate.)
- File is open for writing and must be closed before it can be read.

- Only one record may be read for a variable length file. In this case, the number of records is greater than 1.
- End-of-file (record number specified exceeds number of records in file).
- Variable file has invalid displacement in active file table.
- Invalid character detected in filename.
- Invalid character detected in filetype.

System Action: RC = 100.
Execution of the command terminates.

User Action: Attempt to determine the problem from 'Explanation', above, remedy the condition, and retry the command. Or else retry the command, and if the problem persists, contact installation maintenance personnel.

DMSBSC109S VIRTUAL STORAGE CAPACITY EXCEEDED

Explanation: There is no more space available in the user's virtual machine to successfully complete execution of the command. Subsequent execution of certain CMS commands may result in the same problem.

System Action: RC = 104
The system remains in the same status as before the command was issued.

User Action: Use the CP command DEFINE to increase the size of the virtual machine, IPL CMS again and reenter the command. Or reduce the size of the program and retry.

DMSBSC117S PROGRAM EXCEEDS SOURCE STATEMENT MAX 'nnn'

Explanation: nnn is the maximum number of statements allowed in the program.

System Action: RC = 88
Execution of the command terminates. The system remains in the same status as before the command was entered.

User Action: Reduce the number of source statements and retry.

DMSBSC146S UNEXPECTED BASIC COMPILER REQUEST 'nnn'

Explanation: An unsupported SVC request has been received from the BASIC compiler.

System Action: RC = 88
Execution of the command terminates. The system remains in the same status as before the command was entered.

User Action: Retry. If the error persists, contact your installation maintenance personnel.

DMSBSC147S RUN TIME PACKAGE NOT FOUND

Explanation: The execution time package 'BSCRUN MODULE' could not be found on any of the accessed disks.

System Action: RC = 104.
Execution of the command terminates. The system remains in the same status as before the command was entered.

User Action: Contact your installation maintenance personnel.

DMSBSC204W COMPILER ERROR CODE = 'nnnn'

Explanation: A system error was detected during compilation. A code 'nnnn' specifies the number returned from the compiler, where 'nnnn' is one of the following:

151	(VERBS03)	source pointers were destroyed while in the DIM statement processor.
152	(B$PHASE)	source line is without a beginning line number.
153	(B$PHASE)	source line has been detected without an end-of-line character.
154	(B$PHASE)	source line has been detected with an invalid character.
155	(NUCLEUS)	source pointer was destroyed while in an identifier scan routine.
170	(FORMULA)	the temporary storage counter was below its limit.
171	(FORMULA)	unknown operator (binary).
172	(FORMULA)	unknown operator (unary).
173	(FORMULA)	undefined identifier type.
174	(FORMULA)	undefined delimiter.
175	(FORMULA)	stack overflow.

System Action: RC = 'nnnn'

User Action: Retry, if the error persists contact your installation maintenance personnel.

DMSBSC906T UNEXPECTED RETURN CODE

Explanation: While scanning terminal output from the compiler DMSBSC could not find an end of line character.

System Action: RC = 256
Execution of the command terminates. The system remains in the same status as before the command was entered.

User Action: Retry, if the error persists contact your installation maintenance person.

COMPILATION ERROR MESSAGES

Compilation error messages are issued by the BASIC compiler while the program is being translated or prepared for execution. A line number is printed before any message pertaining to a particular line. In other cases, where it would be helpful in correcting a problem, a line number will also be printed.

Compilation errors can be classified as syntax errors (errors in the construction of a statement), program structure errors (errors in the ordering and relationship of statement lines), or program limit errors.

If any compilation error occurs, the program is not executed. The compiler generally continues to scan the rest of the program for additional errors. If the error involves a program limit, compilation is usually terminated. Only one error per statement is detected for a particular compilation.

The compilation error messages are listed alphabetically below.

ARRAY ALREADY DEFINED

Explanation: A DIM statement has been entered to declare an array which has already been defined, either through use or in another DIM statement.

System Action: Compilation is continued; execution is inhibited.

CHARACTER ARRAY IN MAT

Explanation: An alphameric array name has been specified in a MAT statement. Only numeric array names can be used in MAT statements.

System Action: Compilation is continued; execution is inhibited.

EXPRESSION TCO COMPLEX

Explanation: The line contains an expression requiring too much work space to compile or too many temporary storage locations to compute.

System Action: Compilation is continued; execution is inhibited.

User Action: The user can try deleting unnecessary parentheses and preassigning expressions to variables as a means of eliminating this condition.

FOR/NEXT LOOP INCOMPLETE

Explanation: The program contains at least one incomplete FOR loop.

System Action: Compilation is terminated; execution is inhibited.

FOR/NEXT NESTED INCORRECTLY

Explanation: A NEXT statement does not match the preceding FOR statement.

System Action: Compilation is continued; execution is inhibited.

FOR/NEXT NESTED TCO DEEPLY

Explanation: The program contains more than 15 nested FOR loops.

System Action: Compilation is terminated; execution is inhibited.

FOR/NEXT OUT OF SEQUENCE

Explanation: A NEXT statement appears at a point where no incomplete FOR loop exists.

System Action: Compilation is continued; execution is inhibited.

INVALID ARGUMENT OF DET

Explanation: An argument supplied to the DET function is not a matrix.

System Action: Compilation is continued; execution is inhibited.

INVALID LITERAL CONSTANT

Explanation: The line contains a literal constant for which the boundary characters are missing or not balanced.

System Action: Compilation is continued; execution is inhibited.

INVALID NUMERIC CONSTANT

Explanation: The line contains a numeric constant whose absolute value is greater than 1E+75 or less than 1E-78.

System Action: Compilation is continued; execution is inhibited.

INVALID REDIM SPEC

Explanation: Redimensioning has been attempted in a statement that does not permit redimensioning, or has led to an attempt to change the number of dimensions of the matrix.

System Action: Compilation is continued; execution is inhibited.

INVALID USER FUNCTION

Explanation: A user function has been defined more than once.

System Action: Compilation is continued; execution is inhibited.

MATRIX NOT DECLARED

Explanation: A matrix referenced in a MAT statement has not been declared by a DIM statement or through use.

System Action: Compilation a continued; execution is inhibited.

MATRIX NOT 2-DIMEN

Explanation: A one-dimensional matrix has been referenced in a matrix identity, multiplication, transposition, or inversion operation.

System Action: Compilation is continued; execution is inhibited.

NO. OF DIMENSIONS INVALID

Explanation:

1. The number of subscripts in a reference to a matrix does not correspond to the number of dimensions originally declared for the matrix.

2. A reference to an alphameric array contains two subscripts.

System Action: Compilation is continued; execution is inhibited.

NO. OF DIMENSIONS UNMATCHED

Explanation: The matrices specified in a MAT statement do not have the same number of dimensions.

System Action: Compilation is continued; execution is inhibited.

OBJECT PROGRAM TOO LARGE

Explanation: The object program exceeds the maximum storage space allowed.

System Action: Compilation is terminated; execution is inhibited.

User Action: The user can try combining duplicate source code in GOSUB or user function definitions as a means of eliminating this condition. Program chaining may also provide a solution to this problem.

PROGRAM ERROR. COMPILATION TERMINATED

Explanation: A program-check interrupt has occurred during the compilation process.

System Action: Compilation is terminated; execution is inhibited.

User Action: If the problem persists, the user should contact installation management.

SAME MATRIX FOR RESULT/OPERAND

Explanation: The matrix specified to contain the result of a matrix multiplication, transposition, or inversion operation is the same as an operand matrix of the operation.

System Action: Compilation is continued; execution is inhibited.

SYNTAX ERROR IN EXPRESSION

Explanation: The line does not contain a valid expression where one is expected.

System Action: Compilation is continued; execution is inhibited.

SYNTAX ERROR IN STATEMENT

Explanation: The line contains an error in the construction of the statement.

System Action: Compilation is continued; execution is inhibited.

SYSTEM ERROR HAS OCCURRED

Explanation: A language processor error has occurred during the compilation process.

System Action: Compilation is terminated; execution is inhibited.

User Action: If the problem persists, the user should contact installation management.

TOO MANY ARRAY ELEMENTS

Explanation: The space required for array storage exceeds the maximum allocation.

System Action: Compilation is terminated; execution is inhibited.

TOO MANY FOR/NEXT LOOPS

Explanation: The program contains more than 80 FOR loops.

System Action: Compilation is terminated; execution is inhibited.

TOO MANY IMAGE STATEMENTS

Explanation: PRINT USING and MAT PRINT USING statements reference more than 25 image statements.

System Action: Compilation is terminated; execution is inhibited.

TOO MANY STATEMENT LINES

Explanation: The program contains more than 800 statement lines.

System Action: Compilation is terminated; execution is inhibited.

User Action: Program chaining may provide a solution to this problem.

TOO MANY UNDEFINED LINE NUMBERS

Explanation: An undefined line number is a line number that appears as a reference in the text of a source-program line but does not appear as a line number preceding a line in the program. Up to ten such references are permitted, because, for example, they may precede statements not reached during program execution. When more than ten undefined line numbers have been referenced, however, compilation is terminated.

System Action: Compilation is terminated; execution is inhibited.

TOO MANY VARIABLES OR CONSTANTS

Explanation: The space required to store variables and constants exceeds the maximum allocation.

System Action: Compilation is terminated; execution is inhibited.

User Action: Constants entered in DATA statements and storage for arrays are not held in the area reserved for constants and variables. Therefore, the user can try substituting DATA statements or array elements for constants and variables to circumvent this problem.

COMPILATION EXCEPTION MESSAGES

Certain conditions encountered during compilation are recognized as exceptions by the CALL-OS BASIC language processor but do not cause execution to be inhibited. These conditions are identified and handled as described below.

END SUPPLIED

Explanation: There is no END statement in the program.

System Action: Compilation is continued. The code for an END statement is supplied at the end of the object program.

LINES AFTER END IGNORED

Explanation: One or more program lines follow the END statement of a source program.

System Action: Compilation is continued. The END statement is treated as the last statement in the program.

EXECUTION ERROR MESSAGES

When a program error is detected during program execution, a message is printed and execution is terminated. All messages at run time are preceded by the line number of the statement being executed at the time the error occurred with the exception of those messages where a line number would be irrelevant.

ATTEMPT TO WRITE TO INPUT FILE ON LAST WRITE

Explanation: An attempt was made to write to an input file.

DATA ERROR AT LINE nnnnn

Explanation: A GET or MAT GET statement referring to a program-data file has caused a type or format error in the data at the line identified by line number nnnnn of the program-data file.

DIRECTORY SEARCH FAILED

Explanation: An attempt was made to open a shared file, but either the *Directory was not validated for this user group, the file itself was not available, or the file name was not found in the indicated directory.

END OF DATA

Explanation: A READ statement has been executed with insufficient data in the data table.

END OF FILE

Explanation: A GET or MAT GET operation referring to a data file could not be completed because there was insufficient data in the file.

END OF PROGRAM FILE AT LINE nnnnn

Explanation: A GET or MAT GET operation referring to a program-data file could not be completed because there was insufficient data in the file.

ERROR IN ACS FUNCTION...ARGUMENT TOO LARGE

Explanation: The ACS function has been called using an argument whose magnitude is greater than one.

ERROR IN ASN FUNCTION...ARGUMENT TOO LARGE

Explanation: The ASN function has been called using an argument whose magnitude is greater than one.

ERROR IN COS FUNCTION...ARGUMENT TOO LARGE

Explanation: The COS function has been called using an argument whose short-form magnitude is equal to or greater than 2^{18}pi or whose long-form magnitude is equal to or greater than 2^{50}pi.

ERROR IN COT FUNCTION...ARGUMENT TOO LARGE

Explanation: The COT function has been called using an argument whose short-form magnitude is equal to or greater than 2^{18}pi or whose long-form magnitude is equal to or greater than 2^{50}pi.

ERROR IN COT FUNCTION...INFINITE VALUE

Explanation: The COT function has been called using an argument that causes the cotangent to approach infinity.

ERROR IN CSC FUNCTION...ARGUMENT TOO LARGE

Explanation:The CSC function has been called using an argument whose short-form magnitude is equal to or greater than 2^{18}pi or whose long-form magnitude is equal to or greater than 2^{50}pi.

ERROR IN CSC FUNCTION...INFINITE VALUE

Explanation: The CSC function has been called using an argument that causes the cosecant to approach infinity.

ERROR IN EXP FUNCTION...ARGUMENT TOO LARGE

Explanation: The EXP function has been called using an argument whose magnitude is greater than 174.673.

ERROR IN HCS FUNCTION...ARGUMENT TOO LARGE

Explanation: The HCS function has been called using an argument whose magnitude is greater than 174.673.

ERROR IN HSN FUNCTION...ARGUMENT TOO LARGE

Explanation: The HSN function has been called using an argument whose magnitude is greater than 174.673.

ERROR IN LGT FUNCTION...ARGUMENT ZERO OR NEGATIVE

Explanation: The LGT function has been called using an argument whose value is equal to or less than zero.

ERROR IN LOG FUNCTION...ARGUMENT ZERO OR NEGATIVE

Explanation: The LOG function has been called using an argument whose value is equal to or less than zero.

ERROR IN LTW FUNCTION...ARGUMENT ZERO OR NEGATIVE

Explanation: The LTW function has been called using an argument whose value is equal to or less than zero.

ERROR IN SEC FUNCTION...ARGUMENT TOO LARGE

Explanation: The SEC function has been called using an argument whose short-form magnitude is equal to or greater than 2^{18}pi or whose long-form magnitude is equal to or greater than 2^{50}pi.

ERROR IN SEC FUNCTION...INFINITE VALUE

Explanation: The SEC function has been called using an argument that causes the secant to approach infinity.

ERROR IN SIN FUNCTION...ARGUMENT TOO LARGE

Explanation: The SIN function has been called using an argument whose short-form magnitude is equal to or greater than 2^{18}pi or whose long-form magnitude is equal to or greater than 2^{50}pi.

ERROR IN SQR FUNCTION...NEGATIVE ARGUMENT

Explanation: The SQR function has been called using an argument whose value is negative.

ERROR IN TAN FUNCTION...ARGUMENT TOO LARGE

Explanation: The TAN function has been called using an argument whose short-form magnitude is equal to or greater than 2^{18}pi or whose long-form magnitude is equal to or greater than 2^{50}pi.

ERROR IN TAN FUNCTION...INFINITE VALUE

Explanation: The TAN function has been called using an argument that causes the tangent to approach infinity.

EXPONENTIATION ERROR

Explanation: X Y has been attempted with X=0 and Y=0.

FILE IS ALREADY IN USE

Explanation: An OPEN statement has forced the closing of a file, because the file is identified by the file number specified in the OPEN statement. However, the file name specified in the OPEN statement has been assigned to yet another file, and this OPEN and/or the other file-number entry is for output. For example, the third OPEN statement below would cause this error message to be generated:

```
OPEN 2, 'A', OUTPUT
OPEN 3, 'B', INPUT
OPEN 2, 'B', OUTPUT
```

FILE IS CLOSED OR UNASSIGNED

Explanation: A GET, MAT GET, PUT, or MAT PUT operation has been attempted on an inactive file.

FILE IS FOR INPUT

Explanation: A PUT or MAT PUT operation has been attempted on a file opened as an input file.

APPENDIX C: VM/370 BASIC SAMPLE PROGRAM

RAILROAD TARIFF CALCULATION

A railroad tariff for shipping a commodity between two points depends upon two factors: the rate classification and the rate base. The rate classification is related to the type and quantity of the commodity being shipped. The railroads and the Interstate Commerce Commission establish a minimum weight which qualifies a shipment for a carload rate; any shipment weighing less than this minimum is subject to a less-than-carload rate. Other factors, such as type of packaging and special conditions, are sometimes considered but are ignored in this example.

The rate base, which is also determined by the railroads and the Interstate Commerce Commission, stipulates the charge for traffic in a given direction between two given points. The rate from point 1 to point 2 is not necessarily the same as the rate from point 2 to point 1.

The tariff is calculated by multiplying the appropriate rate base determined from the points of origin and destination by the appropriate carload or less-than-carload rate determined from the weight of the shipment.

In this example only one commodity is used; thus only one set of data is necessary for the origin-destination rate base and for the carload or less-than-carload rate.

STATEMENT OF PROBLEM

Write a BASIC program to read the rate base table, the appropriate minimum weight, and the carload and less-than-carload rates for a given commodity from two disk data files. Read terminal input for individual shipments and calculate the appropriate tariff for each shipment. Print all relevant information.

PROGRAM VARIABLES

Flowchart Notation	BASIC Notation	Meaning
RBASE(IORIG, IDEST)	R(I,J)	Rate base table for the origin-destination combination
IORIG	I1	Integer code for the point of origin IORIG = 1,5
IDEST	I2	Integer code for the destination IDEST = 1,5
COMMOD(ITEM, IN)	C(I,J)	Data on the commodity number, minimum weight, carload rate, and less-than-carload rate
ITEM	I	Index of the table row ITEM = 1,6 for six commodities as test data
IN	I3	Index of the table column entries, coded as follows:
	IN = 1	Master commodity number
	IN = 2	Less-than-carload rate
	IN = 3	Minimum weight to qualify for carload rate
	IN = 4	Carload rate
TARIFF	T	Tariff for shipping the commodity
NUMB	N	Commodity number on the input card
POUNDS	P	Weight of the commodity to be shipped

RATE BASE TABLE AND APPLICABLE RATES

The rate base table and the applicable rates are to be read during program execution from two disk data files. These files must be created and placed on the disk in a preceding step. The program shown below performs the necessary operations. Instructions for listing and executing the program are included in lower case.

```
type rtables basic

90 REM PROGRAM TO LOAD FILES FOR RAIL TARIFF PROGRAM
100 OPEN 2,"FILE1",OUTPUT
110 OPEN 3,"FILE2",OUTPUT
120 DIM A(5,5)
130 MAT READ A
140 MAT PUT 2: A
150 MAT READ A(6,4)
160 MAT PUT 3: A
170 CLOSE 2,3
180 PRINT 'WRAIL COMPLETE'
190 DATA 2.49,3.70,2.72,1.95,3.28,.93,3.03,2.26,1.55,4.25
200 DATA 1.59,3.92,2.94,1.05,3.29,.73,3.08,2.46,1.50,4.30
210 DATA 3.01,6.07,5.73,2.95,5.63
220 DATA 8750,100,24000,55,8790,250,12000,85
230 DATA 8820,125,20000,70,8863,200,20000,85
240 DATA 8885,125,12000,85,8900,70,36000,35
250 END
R;

basic rtables
WRAIL COMPLETE
R;
```

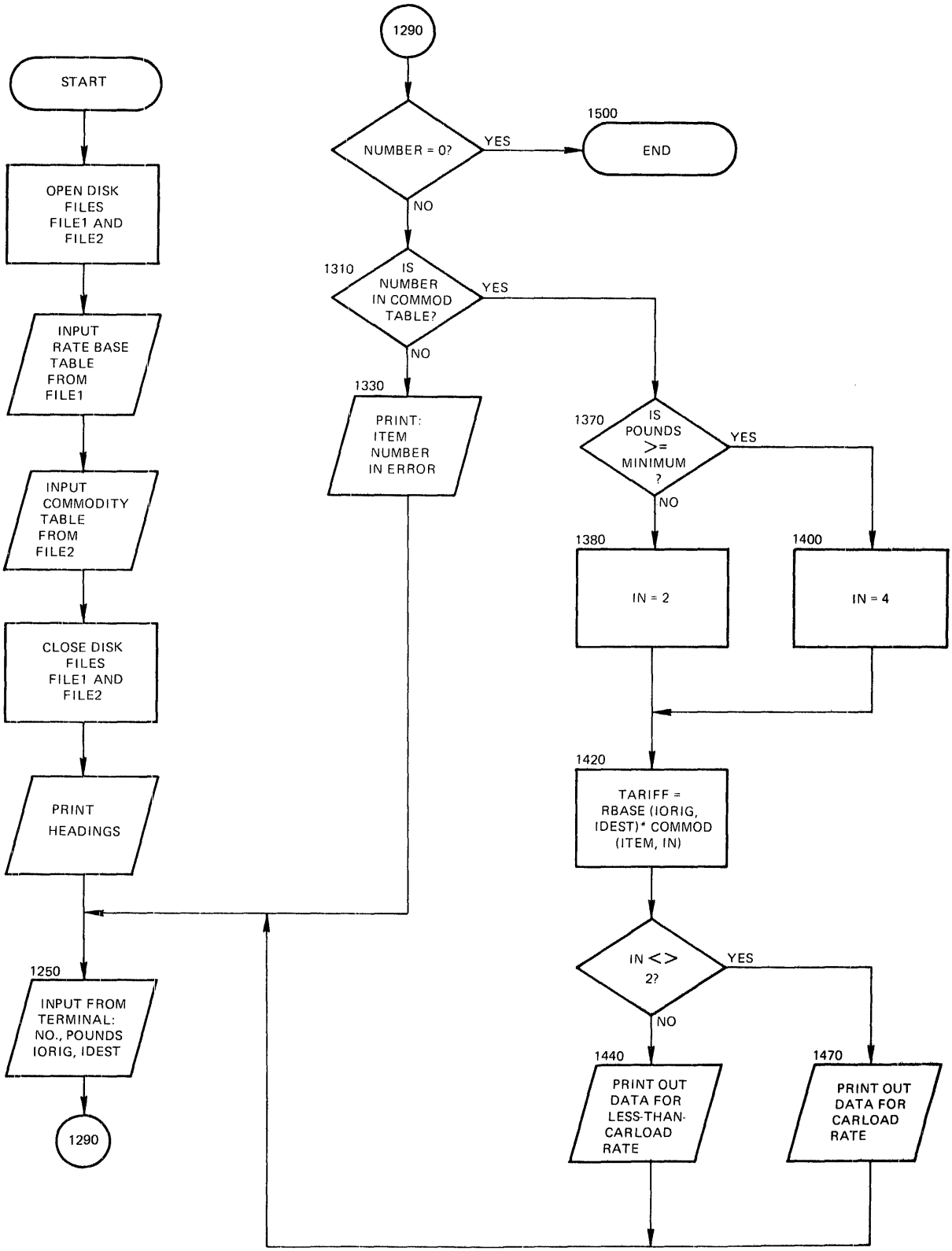
```
RBASE(IORIG, IDEST)

IDEST =    1          2          3          4          5

IORIG =     1         2.49       3.70       2.72       1.95 3.28
            2         0.93       3.03       2.26       1.55 4.25
            3         1.59       3.92       2.94       1.05 3.29
            4         0.73       3.08       2.46       1.50 4.30
            5         3.01       6.07       5.73       2.95 5.63

COMMOD(ITEM, IN)

IN =              1          2          3          4
            NUMBER  LESS THAN  MIN.       CARLOAD
                    CARLOAD                WEIGHT     RATE
                    RATE

ITEM = 1  08750     100.       24000.     .055.
       2  08790     250.       12000.     085.
       3  08820     125.       20000.     070.
       4  08863     200.       20000.     085.
       5  08885     125.       12000.     085.
       6  08900     070.       36000.     035.
```

TEST DATA

Test data appropriate for the railroad tariff calculation program is shown below. One set of four values should be entered each time that the terminal user is asked for input.

Number	Pounds	From	To
08750	20000.	1	2
08790	10000.	1	3
08820	18000.	1	4
08863	18000.	1	5
08885	10000.	2	1
08900	30000.	2	3
08900	40000.	2	4
08885	15000.	2	5
08863	21000.	3	1
08820	21000.	3	2
08790	13000.	3	4
08750	25000.	3	5
08820	20000.	4	1
08863	20000.	4	2
08900	36000.	4	3
08750	24000.	4	5
08790	12000.	5	1
07750	24000.	5	2
08862	20000.	5	3
09900	36000.	5	4

PROGRAM FLOWCHART



PROGRAM LISTING

```
type railtaf basic

1000 REM RAILROAD TARIFF CALCULATION
1010 REM VM/370 BASIC
1020 DIM R(5,5),C(6,4)
1030 REM NEXT TWO STATEMENTS OPEN THE TWO DATA FILES
1040 OPEN 1,'FILE1',INPUT
1050 OPEN 2,'FILE2',INPUT
1060 REM THIS SECTION READS THE DATA FROM THE FILES
1070 REM FILE1 HAS THE RATE BASE TABLE
1080 REM FILE2 HAS THE COMMODITY INFORMATION
1090 FOR I=1 TO 5
1100 FOR J=1 TO 5
1110 GET 1: R(I,J)
1120 NEXT J
1130 NEXT I
1140 FOR I=1 TO 6
1150 FOR J=1 TO 4
1160 GET 2: C(I,J)
1170 NEXT J
1180 NEXT I
1190 CLOSE 1,2
1200 REM PRINT HEADINGS
1210 PRINT 'RAILROAD TARIFF CALCULATIONS'
1220 PRINT
1230 PRINT 'C = CARLOAD RATE'
1240 PRINT 'LC = LESS THAN CARLOAD'
1250 PRINT
1260 PRINT 'ENTER NUMB,POUNDS,ORIG,DEST'
1270 REM TERMINAL USER IS ASKED FOR INPUT
1280 INPUT N,P,I1,I2
1290 IF N=0 GO TO 1500
1300 FOR I=1 TO 6
1310 IF N=C(I,1) GO TO 1370
1320 NEXT I
1330 REM NUMB NOT EQUAL TO ANY C(I,1), NUMB IN ERROR
1340 PRINT 'ITEM NUMBER IN ERROR'
1350 GO TO 1250
1360 REM CHECK FOR MIN. WEIGHT TO DETERMINE RATE
1370 IF P>=C(I,3) GO TO 1400
1380 I3=2
1390 GO TO 1420
1400 I3=4
1410 REM COMPUTE TARIFF
1420 T = R(I1,I2)*C(I,I3)
1430 IF I3<>2 GO TO 1470
1440 PRINT USING 1450 ,C(I,I3),T
1450 : RATE = ###.##LC     TARIFF = ###.##
1460 GO TO 1250
1470 PRINT USING 1480 ,C(I,I3),T
1480 : RATE = ###.##C      TARIFF = ###.##
1490 GO TO 1250
1500 END
R;
```

PROGRAM OUTPUT

```
basic railtaf

RAILROAD TARIFF CALCULATIONS

C = CARLOAD RATE
LC = LESS THAN CARLOAD

ENTER NUMB,POUNDS,IORIG,IDEST? 8750,20000,1,2
 RATE = 100.00LC      TARIFF = 370.00

ENTER NUMB,POUNDS,IORIG,IDEST? 8790,10000,1,3
 RATE = 250.00LC      TARIFF = 680.00

ENTER NUMB,POUNDS,IORIG,IDEST? 8820,18000,1,4
 RATE = 125.00LC      TARIFF = 243.75

ENTER NUMB,POUNDS,IORIG,IDEST? 8863,18000,1,5
 RATE = 200.00LC      TARIFF = 656.00

ENTER NUMB,POUNDS,IORIG,IDEST? 8885,10000,2,1
 RATE = 125.00LC      TARIFF = 116.25

ENTER NUMB,POUNDS,IORIG,IDEST? 8900,30000,2,3
 RATE =  70.00LC      TARIFF = 158.20

ENTER NUMB,POUNDS,IORIG,IDEST? 8900,40000,2,4
 RATE =  35.00C       TARIFF =  54.25

ENTER NUMB,POUNDS,IORIG,IDEST? 8885,15000,2,5
 RATE =  85.00C       TARIFF = 361.25

ENTER NUMB,POUNDS,IORIG,IDEST? 8863,21000,3,1
 RATE =  85.00C       TARIFF = 135.15

ENTER NUMB,POUNDS,IORIG,IDEST? 8820,21000,3,2
 RATE =  70.00C       TARIFF = 274.40

ENTER NUMB,POUNDS,IORIG,IDEST? 8790,13000,3,4
 RATE =  85.00C       TARIFF =  89.25

ENTER NUMB,POUNDS,IORIG,IDEST? 8750,25000,3,5
 RATE =  55.00C       TARIFF = 180.95

ENTER NUMB,POUNDS,IORIG,IDEST? 8820,20000,4,1
 RATE =  70.00C       TARIFF =  51.10

ENTER NUMB,POUNDS,IORIG,IDEST? 8863,20000,4,2
 RATE =  85.00C       TARIFF = 261.80

ENTER NUMB,POUNDS,IORIG,IDEST? 8900,36000,4,3
 RATE =  35.00C       TARIFF =  86.10

ENTER NUMB,POUNDS,IORIG,IDEST? 8750,24000,4,5
 RATE =  55.00C       TARIFF = 236.50

ENTER NUMB,POUNDS,IORIG,IDEST? 8790,12000,5,1
 RATE =  85.00C       TARIFF = 255.85

ENTER NUMB,POUNDS,IORIG,IDEST? 7750,24000,5,2
ITEM NUMBER IN ERROR

ENTER NUMB,POUNDS,IORIG,IDEST? 8862,20000,5,3
```

```
ITEM NUMBER IN ERROR

ENTER NUMB,POUNDS,IORIG,IDEST? 9900,36000,5,4
ITEM NUMBER IN ERROR

ENTER NUMB,POUNDS,IORIG,IDEST? 0,0,0,0

R;
```

READER'S COMMENTS

Title: IBM Virtual Machine Facility/370: BASIC Language Reference Manual

Order No. GC20-1803-1

Please check or fill in the items; adding explanations/comments in the space provided.

Which of the following terms best describes your job?

- ☐ Programmer
- ☐ Manager
- ☐ Operator
- ☐ Instructor
- ☐ Systems Analyst
- ☐ Engineer
- ☐ Mathematician
- ☐ Student/Trainee
- ☐ Customer Engineer
- ☐ Systems Engineer
- ☐ Sales Representative
- ☐ Other (explain below)

Does your installation subscribe to the SL/SS? ☐ Yes ☐ No

How did you use this publication?

- ☐ As an introduction
- ☐ As a reference manual
- ☐ For another purpose (explain) _____
- ☐ As a text (student)
- ☐ As a text (instructor)

Did you find the material easy to read and understand? ☐ Yes ☐ No (explain below)

Did you find the material organized for convenient use? ☐ Yes ☐ No (explain below)

Specific criticisms (explain below)

Clarifications on pages _____

Additions on pages _____

Deletions on pages _____

Errors on pages _____

Explanations and other comments:

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

Trim Along This Line

Trim Along This Line

YOUR COMMENTS PLEASE . . .

This manual is one of a series which serves as a reference source for systems analysts, programmers, and operators of IBM systems. Your comments on the back of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

FOLD FOLD

FIRST CLASS
PERMIT NO. 172
BURLINGTON, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

IBM CORPORATION
VM/370 Publications
24 New England Executive Park
Burlington, Massachusetts 01803

FOLD FOLD

IBM VM/370: BASIC Lang Ref Manual Printed in U.S.A. GC20-1803-1

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)