IBM BASIC

Application Programming: Language Reference

Systems

Program Number 5668-996
Release 1

GC26-4026-0
File No. S370-23

This publication was produced using the
IBM Document Composition Facility
(program number 5748-XX9) and
the master was printed on the IBM 3800 Printing Subsystem.

First Edition (November 1982)

This edition applies to Release 1 of IBM BASIC, Program Product 5668-996, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/370 and 4300 Processors Bibliography, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below; requests for 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, comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California, U.S.A. 95150. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

ABOUT THIS MANUAL

This manual provides reference material on the IBM BASIC language. It presents definitions and examples of IBM BASIC statements and commands.

IBM BASIC is available as a program product that runs under:

- Virtual Machine/System Product—Conversation Monitor System (VM/SP-CMS) Release 1, batch and interactive.

MANUAL ORGANIZATION

In this manual, the following subjects are discussed:

INDUSTRY STANDARDS

The IBM BASIC program product is designed according to the specifications of the following industry standards, as understood and interpreted by IBM as of December 1981:

- American National Standard for Minimal BASIC, ANSI X3.60-1978
- International Organization for Standardization proposed standard ISO Minimal BASIC dp ISO-6373
- European Computer Manufacturers' Association Standard ECMA-55 Minimal BASIC, January 1978

These standards are technically equivalent.

In addition, IBM BASIC has many capabilities not contained in the above standards.

RELATED PUBLICATIONS

Familiarity with the following publications is strongly recommended.

IBM BASIC Application Programming: Guide, SC26-4027

IBM BASIC Application Programming: System Services, SC26-4028

If VSAM file processing is to be used, the following manual is useful:

OS/VS Virtual Storage Access Method: Programmer's Guide, GC26-3838

If BASIC programs are to communicate with Graphical Data Display Manager programs, the following manual is useful:

Graphical Data Display Manager: User's Guide, SC33-0101

CONTENTS

FIGURES

INTRODUCTION

BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code. The elementary capabilities of the language are specified by the American National Standard for Minimal BASIC, ANSI X3.60-1978.

IBM BASIC is a significant extension of Minimal BASIC. It includes many new and advanced capabilities which allow you to produce more powerful and efficient programs. In addition to the programming language, IBM BASIC provides two modes of using the language:

- Interactive mode, which includes an interactive "environment" in which you can create and execute BASIC programs.
- Batch mode, which allows you to separately compile BASIC programs and then execute the compiled programs under direct operating system control. This is similar to other batch language processors such as FORTRAN or COBOL.

The IBM BASIC program product consists of a Processor and a Library. Interactive mode requires both the processor and library. Batch mode requires the processor and library for compilations but only the library for the running of compiled programs.

THE BASIC LANGUAGE

IBM BASIC is a line-oriented language used to generate programs. A program is a sequence of lines containing statements. Each line begins with a unique line number which serves as a label for the first statement contained in the line. Statements are grouped in several categories:

- Declarative
- Control
- Assignment
- Input/output
- Program segmentation
- Exception handling
- Debugging
- Remarks

INTERACTIVE ENVIRONMENT

Interactive IBM BASIC provides the capability to create, edit, debug, and run programs using an interactive terminal. In the interactive environment, the programs themselves can also interact directly with the terminal.

In addition to the BASIC language, the interactive environment provides line number editing facilities and a set of commands. These commands instruct the interactive BASIC processor to perform the following functions:

- Create and edit program lines
- Load, merge, or save programs
- Initiate and control execution of programs

- Display information about IBM BASIC
- Delete files

In the interactive environment, several BASIC statements play a dual role. These statements can be used within programs, as usual, or can be executed immediately to act as commands. With immediate statements, you can perform several desk calculator operations:

- Assignment of values to variables or arrays
- Printing of variables, arrays, or expressions
- Calculations using intrinsic functions and numeric, character, and array operations
- Declaration of immediate variables
- Specification of immediate options

Immediate statements are particularly useful in debugging. They allow you to inspect and modify variables within a program while execution of that program has been suspended.

BATCH ENVIRONMENT

Programs can be compiled either in the interactive environment (with the COMPILE command) or in the batch environment. The compiled programs can also be executed in either the interactive or batch environment.

The batch processor is invoked directly from the host system for each source program file which is to be compiled. It optionally produces an object program file and a listing file.

Programs which are executed in the batch environment do not have an interactive terminal associated with them. Consequently, batch programs do not interact directly with a terminal and do not have the IBM BASIC command and immediate statements available for debugging and program control. In addition, all input/output is from and to internal or external files.

SYNTAX NOTATION

The following conventions are used for syntax notation.

Symbol	Meaning
[ ]	Brackets enclose optional data that can be omitted without causing errors. Example [file-spec] can be specified or omitted in the RUN command.
|	The OR sign separates items for which one choice may be made within a set of data. Example For DISPLAY|INTERNAL|NATIVE, the user may choose only one of the alternatives: DISPLAY, or INTERNAL, or NATIVE.
{ }	Braces indicate a choice of required operands. Choose one alternative from the enclosed set of data. Example For {DISPLAY|INTERNAL|NATIVE}, the user must choose only one of the alternatives: DISPLAY, or INTERNAL, or NATIVE.
...	The ellipsis (...) indicates that the preceding syntactic element may be repeated an arbitrary number of times. Example argument [,argument]... indicates a list of arguments separated by commas.

Note: The syntax notation used in the HELP facility differs from that described above. The HELP syntax notation is described in the HELP SYNTAX panel.

STRUCTURE OF A BASIC PROGRAM

The general syntax of a BASIC program is defined in this chapter.

CHARACTER SET

The IBM BASIC character set is:

Character	Meaning
A - Z	Uppercase Letters
a - z	Lowercase Letters
0 - 9	Digits
	Blank
&	Ampersand
'	Apostrophe (or single quote)
*	Asterisk
@	At Sign
:	Colon
,	Comma
$	Dollar Sign
=	Equal Sign
!	Exclamation Mark
^ or ¬	Exponentiation symbol
>	Greater Than Sign
(	Left Parenthesis
<	Less Than Sign
-	Minus Sign
#	Number Sign
%	Percent Sign
.	Period
+	Plus Sign
"	Quotation mark (or double quote)
)	Right Parenthesis
;	Semicolon
/	Solidus or Slash
_	Underline

Notes:

1. Uppercase and lowercase letters are equivalent in a BASIC program, except within character string.

2. The processor uses the asterisk (*) and question mark (?) characters as terminal prompts.

IDENTIFIERS

Identifiers name variables, arrays, functions, subprograms and line labels.

The names of variables, arrays, functions, and line labels may contain up to 40 characters. Subprogram names (see SUB and CALL statements) may contain at most seven characters.

The first character of an identifier must be a letter, which may be followed any of the 26 letters of the alphabet, the 10 digits, and the underline character (_). Letters may be uppercase or lowercase.

The final character of an identifier may be the number sign (#), the percent sign (%), or the dollar sign ($). Special meanings provided by the three characters are discussed under "Variables" on page 17.

Examples

```
ALPHA$
Alpha$
alpha_betic$
alpha___betic$
DEC_NUM
INT_num%
decNum#
```

A given identifier may name:

- a variable,
- an array,
- or a function.

but not more than one of these in a program unit.

Within a program unit, the same identifier may be used as:

- a variable or array name,
- a line label,
- or a subprogram name (SUB statement).

because context always determines which interpretation is to be used.

A program unit is either a main program or a subprogram. Each program unit is a distinct entity in that identifiers used to name variables, arrays, and user-defined functions are local to the program units in which they occur; that is, they may be used to name different objects in different program units. Identifiers used to name subprograms are global to the entire program; that is, they name the same subprogram wherever they occur.

LANGUAGE STATEMENTS

BASIC source language statements contain line numbers, optional line labels, keywords, identifiers, and expressions.

LINES AND LINE NUMBERS

An IBM BASIC program is made up of a series of statement lines. Each line starts with a unique number. The smallest line number allowed is 1 and the largest 9999999.

Line numbers provide labels for statements.

Example

```
100 IF A=B THEN 300
.
.
.
300 LET B=C
```

The statement at line 100 directs processing to bypass all of the instructions between lines 100 and 300 if the value of the variable A is equal to the value of the variable B. Line (or statement) 300 becomes a label which is the object of the IF statement.

LINE LABELS

A line label is a statement identifier followed by a colon (:). It is declared by its appearance after a line number.

Only one line label may be declared for any one line number.

Line labels may be up to 40 characters long, the first of which must be alphabetic (A-Z). The remaining characters may be either alphabetic, numeric, or the underline character.

Example

```
200 FIRST_CHOICE: LET A=B
```

In this example, FIRST_CHOICE is the line label.

RESERVED WORDS

Reserved words are words you cannot use as identifiers.

Unless your organization has customized the reserved word list, all of the keywords in IBM BASIC as distributed are reserved words.

Check with your system administrator for the reserved words used by your organization.

KEYWORDS

Following the line number, and the optional line label, the line usually continues with one or more BASIC statements. A statement usually begins with a keyword. Each keyword in BASIC has a specific meaning. Some keywords are optional and are so noted in the descriptions of those statements.

The initial keyword of a statement indicates the action to be performed by the statement (READ, WRITE, etc.).

Keywords may be spelled using either lowercase letters, uppercase letters, or mixed uppercase and lowercase letters.

Example

```
LET
Let
let
```

are equivalent.

Keyword List

The following is a partial list of IBM BASIC keywords in alphabetic order. (The complete list includes the names of the intrinsic functions and predefined subprogram names. Intrinsic function names are listed in "Intrinsic Functions" on page 34. Predefined subprogram names are listed in "Predefined Subprogram Names" on page 91.)

ACCESS
AND
APPEND
AT
ATTN

BASE
BEGIN
BOTTOM
BREAK

CALL
CASE
CAUSE
CHAIN
CLOSE
COLLATE
COM
COMMON
CONTINUE
CONV

DATA
DEBUG
DECIMAL
DEF
DEFDBL
DEFINT
DEFSNG
DELETE
DIM
DISPLAY
DO
DUPKEY
DUPREC

ELSE
END
ENDPAGE
EOF
EQ
ERROR
EXIT

FIELDS
FILES
FIPS
FIXED
FLAG
FNEND
FONT
FOR
FORM

GE
GET
GO
GOSUB
GOTO
GT

IF
IGNORE
IMAGE
INPUT
INTEGER
INTERNAL
INVP
IOERR

KEY
KEYED

LE
LEFT
LENGTH
LET
LINE
LINPUT
LOOP
LPREC
LT

MARGIN
MAT

NATIVE
NE
NEWPAGE
NEXT
NOFIPS
NOKEY
NONE
NOREC
NOT

OFF
OFLOW
ON
OPEN
OPTION
OR
ORGANIZATION
OUTIN
OUTPUT

PAGE
PAGEOFLOW
PAUSE
POINTER
POS
PRINT
PROMPT
PRTZO
PUT

RANDOMIZE
RD
READ
REC
RECORD
RECORDS
RELATIVE
REM
REREAD
RESET
REST
RESTORE
RETRY
RETURN
REWRITE
RIGHT

SCRATCH
SEARCH
SELECT
SEQUENTIAL
SKEY
SKIP
SOFLOW
SPREC
STANDARD
STEP
STOP
STREAM
SUB
SUBEXIT
SYSTEM

TAB
THEN
TO
TOP
TRACE
TYPE

UFLOW
UNTIL
USE
USING

VARIABLE

WHILE
WRITE

ZDIV

Except for the following, keywords may not be abbreviated.

- COM may be used in place of COMMON
- REC may be used in place of RECORD
- PAGE may be used in place of NEWPAGE

Rules for Keywords Removed from the Reserved Word List

If your organization has removed keywords from the IBM BASIC reserved word list, the rules regarding treatment of keywords are different from those given above; the modified rules are given in the following paragraphs.

If a BASIC keyword has not been removed from the reserved word list, the keyword cannot be used as an identifier.

Example

```
LET LET = 2

LET = 2
```

neither is accepted if LET is included in the reserved word list.

However, if a keyword has been removed from the reserved word list, it can be used as an identifier, even though it is still a BASIC keyword. In that case, both of the LET statements in the preceding example are accepted as written:

- In the first LET statement, the first word LET is interpreted as a keyword, and the second is interpreted as an identifier.
- In the second LET statement, the word LET is interpreted as an identifier.

Some keywords may be ambiguous if they are not reserved; therefore, whenever an ambiguity is detected, the keyword interpretation is used.

Example

```
100 REM = 3

200 IMAGE: X=Y

300 X = SIN(Y)
```

In these examples:

- Statement 100 is a REM statement, not an assignment statement.
- Statement 200 is an IMAGE statement, not an assignment statement with a statement label named IMAGE.
- Statement 300 is a reference to the intrinsic function SIN, not an implicit declaration of a numeric array SIN.

However, if a keyword is deleted from the reserved word table and its first use in the program is a declaration in a COM, DIM, or DEF statement, an ambiguous usage is treated by the program as an identifier. That is, in the previous example, if SIN is declared as an array in a COM statement before the reference in statement 300, then statement 300 is a reference to an array named SIN.

MEANINGFUL SPACES OR BLANKS

Spaces (blanks) cannot appear within:

- line numbers
- keywords
- identifiers
- numeric constants

Example

```
BOT TOM
```

is not acceptable.

Spaces are optional between the following keywords. GOTO and GO TO mean the same thing, GOSUB and GO SUB mean the same thing.

When the presence of delimiting characters delimits keywords or identifiers, the keywords or identifiers can be coded with or without delimiting spaces. Delimiting characters are:

```
+ - * / = ( ) " ' ¬ (or ^)
```

All the following assignment statements are accepted and processed in the same manner:

Example

```
100 LET TOTAL = VALA + VALB+VALC
100 LET TOTAL = VALA+VALB + VALC
100 LET TOTAL = VALA + VALB + VALC
```

Any keyword appearing in a program must be preceded by a space or other delimiting character and, if not at the end of a line, followed by a space or other delimiting character.

Example

```
FORI=1 TO10
```

is not a correct FOR statement.
A space must follow the
words FOR and TO, as follows:

```
FOR I=1 TO 10
```

Spaces may optionally precede or follow the equal sign (=).

Example

```
FOR I =1 TO 10
FOR I= 1 TO 10
FOR I = 1 TO 10
```

are all equivalent.

Spaces appearing in quoted characters constants (character constants enclosed in quotes) are counted as part of the constant; spaces appearing outside the quotation are not considered part of the constant. Spaces either preceding or following an unquoted character constant (only allowed in DATA statements and responses to INPUT statements) are not considered part of the character string.

Example

```
LET ALPHA$ = "LAST YEAR"
```

The space between T and Y is part of the string

```
DATA      LAST YEAR
```

The space between T and Y is part of the string; the spaces that precede L and follow R are not part of the string

STATEMENTS AND THEIR CATEGORIES

IBM BASIC statements can be considered as belonging to one of the following categories:

Imperative Statement Causes an unconditional action to occur.

Conditional Statement Tests a condition to determines which of two or more alternative paths of execution are to be followed.

Declarative Statement Specifies characteristics of the program in general and thus influences the entire program unit in which it appears.

Continuation of Statements

Statements may be continued from one line to the next by placing an ampersand (&) as the last nonblank character of each line to be continued, and beginning the next line with an ampersand as the first nonblank character.

Example

```
100 PRINT A,B,C,D,E(11,12),F,G,H,I,J,K,L&
  & ,M,N
```

is equivalent to:

```
100 PRINT A,B,C,D,E(11,12),F,G,H,I,J,K,L,M,N
```

A line can be continued in any line position where a space might normally appear, except within a character constant. Line numbers and line labels are not allowed on continuation lines.

Continuations are not allowed within REM (remarks) statements. Lines containing trailing comments may be continued, but the comment may not be continued.

Example

```
100 A = B + ! THIS IS A COMMENT &
  & C
```

is functionally equivalent to:

```
100 A = B + C ! THIS IS A COMMENT
```

There is no limit to the number of continuations, except for the amount of storage available to the entire program.

Multiple Statements Per Line

The colon (:), when not within a quoted character string or parentheses, and when not used to signify a label, and when not used in a file I/O or IMAGE statement, indicates the end of a statement when another statement begins on the same line.

Example

```
100 LET A = 5: LET B = 6: LET C = 10
```

is functionally equivalent to three separate LET statements:

```
100 LET A = 5
110 LET B = 6
120 LET C = 10
```

If a statement normally could be expected to end in a colon, then the presence of a second colon is recognized as the indication that another statement is present on the same line.

Example

```
100 PRINT #3:: PRINT #3: "THE ANSWER IS", A
```

is functionally equivalent to two separate PRINT statements:

```
100 PRINT #3:
110 PRINT #3: "THE ANSWER IS", A
```

where the first colon is part of the syntax of the PRINT #3: statement, and the second colon indicates that another statement begins on that line.

The statements EXIT, IMAGE, FORM, and SUB must be the first statement on a line, because they must have a line number or label to be used for reference.

The DATA, EXIT, FORM, IF, IMAGE, and REM statements must be last on a line. This requirement is made to maintain compatibility with other BASICs and to avoid ambiguous syntax.

In a REM statement, or in any statement containing an exclamation mark remark (!), no other statement can occur on the same line, since all characters following the remark are taken to be remarks.

Similarly, wherever an unquoted character string may occur, such as in a DATA statement, another statement may not follow, because of the possible conflict of interpretation.

When the line number of a multistatement line is used in a GOTO or GOSUB statement, it always refers to the first statement on that line.

Multiple statements per line are not allowed in immediate statements.

Colons are used to separate statements within statement lists appearing in IF statements. When an IF statement ends with a statement list (as opposed to a line number reference or a line label reference), all of the remaining statements on the line are considered part of the IF statement list.

COMMENTS

Comments inserted at intervals make programs easier to understand and their logic easier to follow. IBM BASIC allows comments in two forms: the exclamation mark comment, and the REM comment.

EXCLAMATION MARK COMMENTS

An exclamation mark (!) specifies that the balance of the data on the current line is a comment and not to be interpreted, that is, the data is to be displayed in the program listing and no other action taken.

Example

```
100 LET DEPOSITS = 90.10 + 50.00 + 85.00  !TOTAL DEPOSITS
110 LET CHECKS   = 12.50 + 16.00 + 27.00  !TOTAL CHECKS
120 LET BALANCE  = OLDBAL+DEPOSITS-CHECKS !NEW BALANCE
```

Through the comments, the purpose of these three statements is clearly documented.

The exclamation mark may not appear as a trailing comment on an IMAGE, DATA, or FORM statement.

REM STATEMENT COMMENTS

The keyword REM (for remarks) is used for comment lines. REM specifies that data on the entire current line is a comment and not to be interpreted; that is, the data is to be displayed in the program listing and no other action taken.

Example

```
100 LET ABC = PARTA + PARTB
110 REM TOTAL THE PARTS
```

has exactly the same meaning as the following line:

```
100 LET ABC = PARTA + PARTB !TOTAL THE PARTS
```

COMMENT CONTINUATION NOT ALLOWED

Comments cannot extend beyond an end of line. If more room is required for a comment than the current line allows, another REM, or exclamation mark permits continuation of the comment at the beginning of the next line.

Example

```
100 LET DEPOSITS = 90.10 + 50.00 + 85.00 !TOTAL
110 !DEPOSITS FOR CURRENT MONTH
```

STATEMENT BLOCKS

Certain statements are logically grouped into statement blocks. Each block serves a separate and distinct purpose. The statement blocks are:

- User-Defined Functions—described in "User-Defined Function Statements" on page 77
- Loop Blocks—described in "Loop Control Statements" on page 62
- IF Blocks—described in "IF Blocks" on page 64
- SELECT/CASE Blocks—described in "SELECT Blocks" on page 66

PROGRAM UNITS

A program may be divided logically into a number of program units; a main program and one or more subprograms.

Each program unit establishes a separate scope of identifiers. The same identifier may be used in different program units to name different items.

Statements within a program unit may not refer to any variable, array, line label, line number, or function (other than intrinsic functions) defined externally to that program unit.

Program units are described in "Subprogram Statements" on page 78.

CONSTANTS, VARIABLES, AND ARRAYS

Data can be constants, variables, or arrays. To reference data, a constant, a variable name, or an array name may be specified.

All data—whether a constant, a variable, or an array—is divided into two classes: numeric and character. There are two types of numeric data: integer or decimal. Character data has the character type.

CONSTANTS

A constant, as the name implies, is a piece of data whose value will not and cannot be changed during processing. There are two types of constants: numeric and character.

NUMERIC CONSTANTS

Numeric (also referred to as arithmetic) constants can be represented in two forms: integer and decimal.

Integer Constants

Integer constants are made up of whole numbers. These numbers can be either positive or negative and can range from -2,147,483,648 to +2,147,483,647.

Periods and commas cannot appear within integer constants.

Example

```
15
+365
-123
1429
```

INTERNAL REPRESENTATION OF INTEGER CONSTANTS: Integer Constants can contain up to 10 digits, with a maximum value of +2,147,483,647 and a minimum value of -2,147,483,648.

They are stored internally as shown in Figure 1.

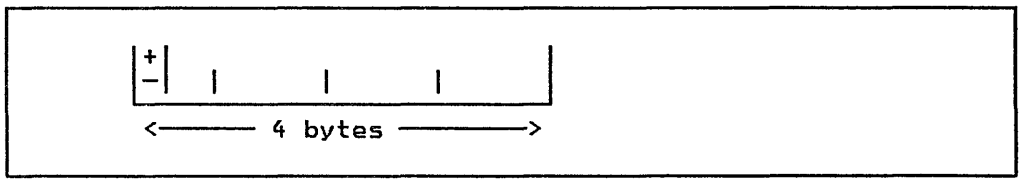


Figure 1. Integer Data—Internal Representation

Integer constants are stored internally as full word (32-bit) two's complement values.

Decimal Constants

Decimal constants can be either fixed-point or floating-point constants.

Decimal data can contain up to 17 digits.

FIXED-POINT DECIMAL CONSTANTS: Fixed-point decimal constants can be either positive or negative in value and must include a decimal point.

Example

```
.15
+3.65
-123.
```

FLOATING-POINT DECIMAL CONSTANTS: Floating-point decimal constants allow the representation of very large or very small numeric values.

Floating-point constants can be either positive or negative in value. A floating-point decimal constant can be written as either an integer or decimal fixed point constant, followed by the letter E, followed by an integer constant.

Example

```
5.0E+6
5E6
+5E06
```

Each of these examples represents the number 5 million, expressed as 5 times 10 to the sixth.

Each of these examples specifies that the decimal point is to be moved right the number of places indicated after the letter E (that is, 5 is to be multiplied by that power of 10). The number before the E is referred to as the mantissa. The number after the E is referred to as the exponent.

Just as easily, very small number values can be presented by saying, "move the decimal left."

Example

```
5.0E-6
5E-06
+5E-6
```

all represent .000005, or 5 times 10 to the minus sixth.

For floating-point, the mantissa is normalized to an implicit decimal point to the left of the leftmost significant digit. The exponent can be in the range -75 to +75.

Therefore, the largest absolute value is:

```
.99999999999999999E+75
```

and the smallest absolute value is:

```
1E-75
```

INTERNAL REPRESENTATION OF DECIMAL CONSTANTS: Decimal data is stored internally as three words (12 bytes) as shown in Figure 2 on page 16.

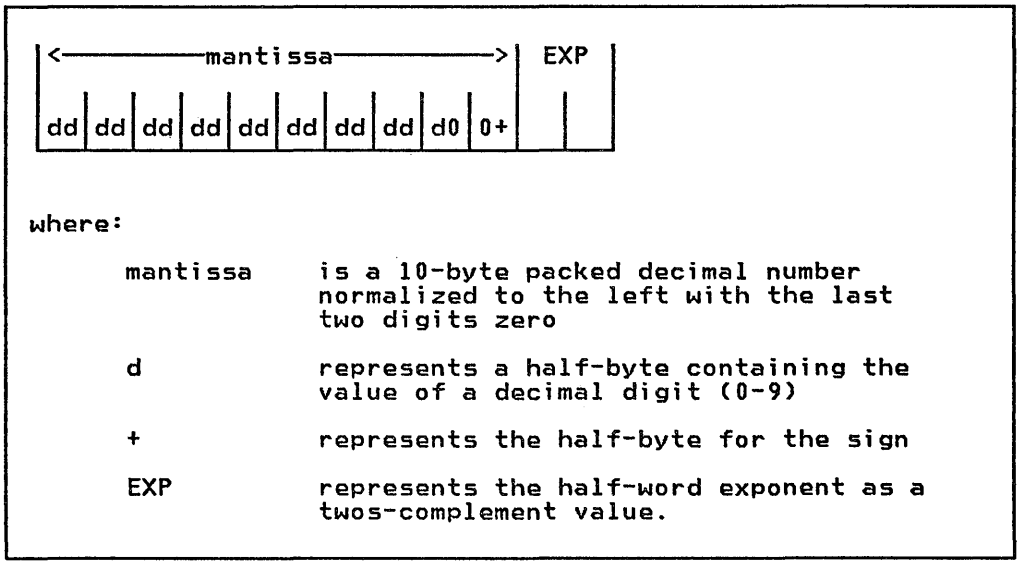


Figure 2. Decimal Data-Internal Representation

CHARACTER CONSTANTS

Character constants are strings of characters. They can be used, for example, to create headings and subheadings for reports. They are normally enclosed within quotes (called delimiters). These delimiters may be omitted in DATA statements and in the responses to INPUT statements under specific rules which are discussed in the sections dealing with the DATA and INPUT statements.

Character constants may be specified using either single or double quotes, but not both, for the same constant.

Example

If this constant is specified:

```
'The sentry shouted, "Halt!"'
```

the value is:

```
The sentry shouted, "Halt!"
```

with double quotes around the word Halt.

If this constant is specified:

```
"The sentry shouted, 'Halt!'"
```

the value is:

```
The sentry shouted, 'Halt!'
```

with single quotes around the word Halt.

If the delimiting quote mark appears within the string, it must appear doubled, with no intervening spaces. Thus, the last example could be specified:

```
'The sentry shouted, ''Halt!'''
```

A character constant may be null, that is, contain no characters. The null string (that is, a string of length zero) may be

specified with either contiguous single quotes ('') or contiguous double quotes ("").

VARIABLES

Variables are data items whose values may be changed. Variables may take two forms: simple, referring to a single item, or subscripted, referring to one item (or member) of an array or group of data.

A variable name is an identifier for a set of data which may change values during processing, hence the name variable.

Variable names cannot exceed 40 characters in length.

NUMERIC VARIABLES

Numeric variables:

- Are integer or decimal type.
- Are specified explicitly by type through the INTEGER or DECIMAL statements.
- Are specified implicitly by type through the integer suffix % or the decimal suffix # on the variable name.
- Are in error if these specifications conflict.
- May not have names ending with $.
- Are initialized to zero (0).

The type (decimal or integer) of numeric variables may be specified by DECIMAL and INTEGER statements. These statements declare the type of a variable according to the variable name or the first letter of the name.

Example

```
100 INTEGER (A-C),INT1
```

defines all numeric variables with names beginning with A, B, or C as type integer. The variable INT1 is also type integer.

If a variable name does not end with # or % and the name or first letter of the name is not specified in a DECIMAL or INTEGER statement, the variable is assigned decimal type by default.

Variable names ending with # or % may not be assigned a contradictory type. For example, DECIMAL X% is an error, INTEGER X% is redundant but acceptable. Also, the # and % typing overrides first character typing (through DECIMAL or INTEGER statements).

Example

```
110 INTEGER(A-C)
120 ALPHA#=BETA
```

In this example ALPHA# is given decimal type even though it begins with one of the letters declared in the INTEGER statement. BETA is typed integer.

Note that the characters % and # are considered as part of the 40 characters when determining variable name size limits.

If a numeric variable name ending with # or % duplicates another numeric variable name in all character positions except the final # or % and the two names have the same type, then they refer to the same variable. For example, A#, A%, and A may all be used in a

program. If A is typed decimal (either by a DECIMAL statement or by default) then A and A# represent the same variable. However, if A is typed integer, A and A% refer to the same variable.

Each time execution of a program unit (main program or subprogram) is begun, all numeric variables local to the program unit (not in COMMON and not parameters) are initialized to zero. COMMON variables are initialized to zero when the first program unit using COMMON is encountered.

Integer variables contain 32-bit two's complement values as described under "Integer Constants" on page 14.

Decimal variables contain 17 digit values with a decimal point and power of ten exponent. The internal representation and range of decimal values is discussed under "Decimal Constants" on page 14.

CHARACTER VARIABLES

Character variables:

- Must have names ending with $.
- Contain strings which are variable in length.
- Have a maximum length which is either declared explicitly in a DIM or COMMON statement, or is the IBM supplied default, (18, but check with your system administrator).
- May have a maximum length of 32767 characters.
- Are initialized to a null (zero length) string.

All character variable names must end with the character $ and may be up to 40 characters in length, including the dollar sign.

Character variables contain character strings of varying length. In other words, the value of a character variable does not necessarily always have the same length.

Each character variable has associated with it a current length. Each character variable also has a maximum length. The maximum length is the maximum value possible for the current length. The maximum length may be declared with the DIM or COMMON statements or, if not declared, defaults to a value determined by your system administrator (18 is the IBM supplied default). The maximum value that may be declared is 32,767.

Example

```
110 A$ = "STRING"
120 A$ = "BIGGER STRING"
```

In the above example, at line 110 A$ is assigned a string of length 6, and at line 120 the same variable is assigned a string of length 13.

Example

```
100 DIM ADDRESS$*30
```

In this example, the "*30" declares the variable ADDRESS$ to have a maximum length of 30.

When execution of a program is initiated, for example, with the RUN command, and before any statements are processed, IBM BASIC initializes all character variables to the null string (sets their current lengths to zero).

ARRAYS

An array is a collection of data items (elements) that is referred to by a single name. Only data items of the same type (character, decimal, or integer) and, in the case of character, the same maximum length can be grouped together to form an array.

Arrays can have from one to seven dimensions. A one-dimensional array can be thought of as a vector of successive data items.

REFERENCES TO ARRAY ELEMENTS (SUBSCRIPTS)

To refer to a single element in the array you must be able to specify that element. In order to do so, you must provide a subscript for each dimension of the array.

A subscript can be any valid arithmetic expression whose rounded integer value is equal to or greater than zero. A set of subscripts is specified within parentheses and separated by commas after the array name to which they refer.

The parentheses and the enclosed term(s) are referred to as a subscript.

Example

```
ARA$(3,3,3)
```

The subscript reference (3,3,3) locates a specific element within the 3-dimensional array named ARA3$.

Reference to an element in a one-dimensional array requires that you specify the array name (ARA1$ for example) followed by the desired array position enclosed in parentheses. If you want to move the third value of ARA1$, assuming OPTION BASE 0, to the variable VAR$, you can use this statement;

```
100 LET VAR$=ARA1$(2)
```

In multidimensional arrays, the rightmost subscript varies most rapidly. If you define ARA2$ in this manner:

```
100 DIM ARA2$(10,20)
```

a two-dimensional array is generated, and (assuming OPTION BASE 1) you can refer to any one of the two hundred locations within ARA2$ by using the proper subscript. 200 LET statements could refer to the array sequentially, one location at a time, in the same order that a MAT statement would use:

```
100 LET VARA1$=ARA2$(1,1)
110 LET VARA2$=ARA2$(1,2)
120 LET VARA3$=ARA2$(1,3)
.
.
.
1990 LET VARA199$=ARA2$(10,19)
2000 LET VARA200$=ARA2$(10,20)
```

Subscript Boundaries

The lower boundary of each subscript is determined by the OPTION BASE in effect. If OPTION BASE 1 is in effect the lower boundary of a subscript is one (1). If OPTION BASE 0 is in effect the lower boundary of a subscript is zero (0). Upper subscript boundaries are dependent upon the current dimensions of the array. These dimensions are initially set by DIM or COMMON statements or by implicit defaults (see "Explicit Dimensioning of Arrays" on page 21 and "Implicit Dimensioning of Arrays" on page 22), but may be dynamically changed by redimensioning (see "Redimensioning" on page 23). Redimensioning must not increase the array size to

exceed the explicitly or implicitly defined number of elements. The intrinsic function SIZE may be used to determine the dimensions of an array or the number of its elements (see "Intrinsic Functions" on page 34).

Subscripts may be any numeric expression. If the result of the expression is decimal with a fractional part, it is rounded and converted to an integer value. Thus, A(2.5) is equivalent to A(3).

If a subscript value is outside of the dimension's range (less than the lower boundary or greater than the upper boundary), an exception is generated when the illegal array reference is attempted (see "ON Condition Statement" on page 203).

BASE INDEXING

Array dimensioning is based on the selection of BASE 0 indexing or BASE 1 indexing. The OPTION statement specifies the base; the default is BASE 0.

If base 0 indexing is specified, the array is referenced starting with zero, A(0), A(1), A(2), A(3), etc. If base 1 indexing is specified, the reference starts with one, A(1), A(2), A(3), A(4), etc.

Example

DIM A(10)

When OPTION BASE 0 is in effect is an 11-element array

When OPTION BASE 1 is in effect is a 10-element array

Figure 3 shows how a one-dimensional array named A, with OPTION BASE 0 in effect, is laid out in storage.

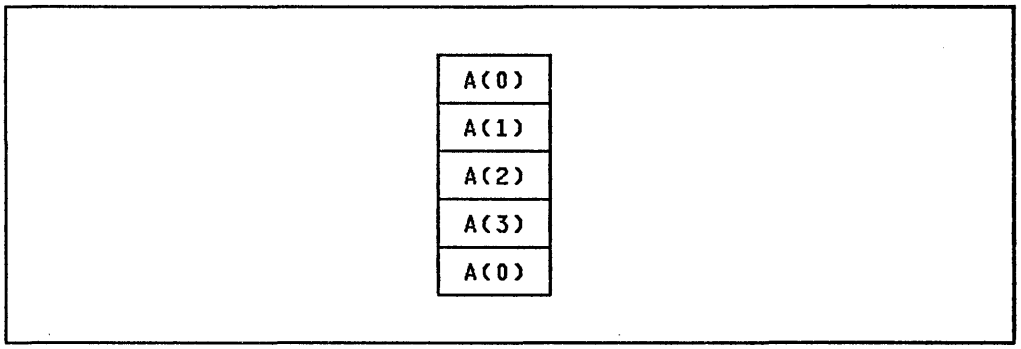
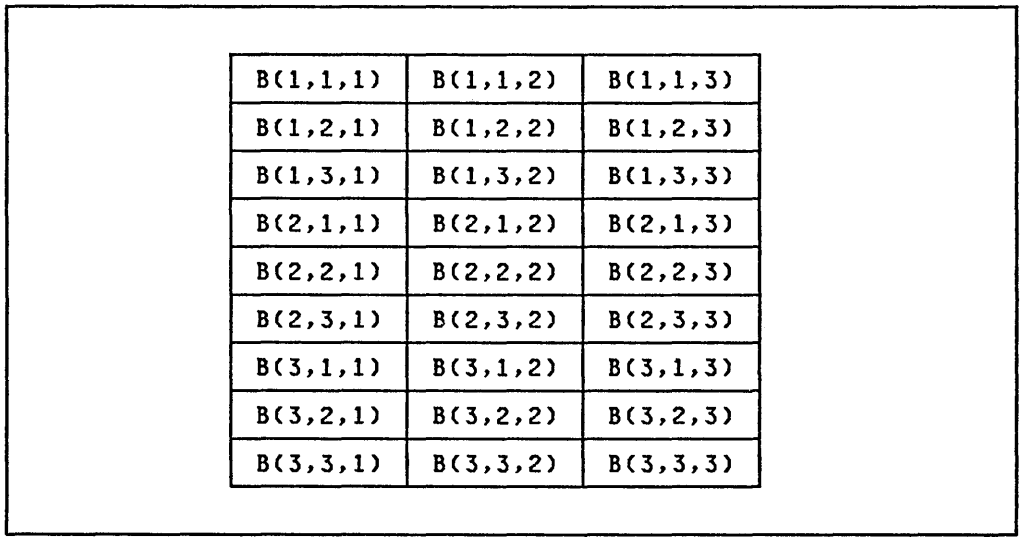


Figure 3. One-Dimensional Array References—BASE 0 Indexing

Figure 4 shows how references to a three-dimensional array Named B, with OPTION BASE 1 indexing, would appear in storage.



B(1,1,1)	B(1,1,2)	B(1,1,3)
B(1,2,1)	B(1,2,2)	B(1,2,3)
B(1,3,1)	B(1,3,2)	B(1,3,3)
B(2,1,1)	B(2,1,2)	B(2,1,3)
B(2,2,1)	B(2,2,2)	B(2,2,3)
B(2,3,1)	B(2,3,2)	B(2,3,3)
B(3,1,1)	B(3,1,2)	B(3,1,3)
B(3,2,1)	B(3,2,2)	B(3,2,3)
B(3,3,1)	B(3,3,2)	B(3,3,3)

Figure 4. Three-Dimensional Array References—BASE 1 Indexing

Note: A series of values assigned to a multidimensional array always fills that array with the rightmost subscript varying most rapidly.

EXPLICIT DIMENSIONING OF ARRAYS

Arrays can be dimensioned explicitly by either the DIM or COMMON statements.

When an array is dimensioned explicitly, both the number of dimensions and the upper bounds of each dimension are specified in the DIM or COMMON statement. For example, if BASE 1 indexing is in effect:

```
DIM ARRAY(20,4)
```

dimensions a numeric array named ARRAY to 20 rows and 4 columns.

The maximum size of an array is limited only by the virtual storage available.

If an array is explicitly dimensioned more than once in a program unit, an error message is printed and all declarations of the array except the first are ignored.

If an array is explicitly dimensioned with an upper bound of zero in a program unit with OPTION BASE 1, an error message is printed and the array declaration is ignored.

Numeric Arrays

Numeric arrays are typed (decimal or integer) in the same manner as numeric variables:

- The DECIMAL and INTEGER statements can declare specific array names or specific beginning letters of array names.
- The characters # and % at the end of array names indicate decimal and integer, respectively, and override declarations in DECIMAL and INTEGER statements.

- If a numeric array name ending with # or % duplicates another array name in all character positions except the final # or %, and the two arrays have the same type, they refer to the same array and only one of the two may be dimensioned explicitly.
- Integer arrays contain values as described for integer variables, and decimal arrays contains values as described for decimal variables (see "Numeric Variables" on page 17).
- Numeric arrays are initialized to zero.

Character Arrays

Character arrays follow much the same rules as character variables:

- Character array names must end with the character $.
- All elements of a character array are assigned the same maximum string length either by explicit declaration in a DIM or COMMON statement or by default. The maximum length is 32767. The default maximum is determined by your system administrator. (18 is the IBM-supplied default.)
- Each element of a character array has a current length which is not necessarily equal to the current length of other elements.
- When processing begins, the current length of all character array elements is set to zero (the null string).

Example

```
100 OPTION BASE 1
110 DIM NAMES_OF_MONTHS$(12)*9
```

This declares a one-dimensional character array of 12 elements with each element having a maximum length of nine characters.

After an array has been explicitly dimensioned, it cannot be explicitly dimensioned a second time by another DIM or COMMON statement anywhere in the program unit.

IMPLICIT DIMENSIONING OF ARRAYS

If the first usage of a name in a program is as an array, but the array is not dimensioned in a DIM or COMMON statement, implicit dimensions are assumed.

The upper bound of each implicit dimension is 10; the lower bound is 0 or 1, depending on OPTION BASE 0 or 1.

The number of implicit dimensions depends on the number of dimensions in the subscript reference.

If the reference has no subscripts, for example, in a MAT assignment statement, two dimensions are assumed. (The MAT functions DOT, AIDX, DIDX and the intrinsic function SRCH are exceptions to this rule; they assume one-dimensional arrays.)

An array can be declared implicitly by using the array name in a context where only an array name is permitted, as in an assignment statement with subscripts. For example, assuming OPTION BASE 0 is in effect:

- Reference with subscripts

  ```
  100 LET ARAX(4,5)=10
  ```

  establishes ARAX as a two-dimensional array, each dimension containing 11 elements.

- MAT statement

  ```
  110 MAT A=(15)
  ```

  establishes A as a two-dimensional array, each dimension containing 11 elements.

- MAT name in an input or output list

  ```
  100 PRINT USING 120:MAT ARAY
  ```

  establishes ARAY as a two-dimensional array, each dimension containing 11 elements.

- As the argument of a function which requires an array parameter

  ```
  100 IF XDT(K)=0 THEN 200
  ```

  establishes XDT as a one-dimensional array containing 11 elements.

Assuming BASE 0 indexing, when no DIM or COMMON statement specifies an array named A, the statement:

```
A(3) = 50
```

establishes a one-dimensional array containing 11 elements. Element A(3) is the 4th element and has a value of 50.

Similarly, when neither a DIM nor a COMMON statement specifies an array named ARRAY, the statement:

```
ARRAY(10,4) = 7.123
```

establishes a two-dimensional array containing 11 rows and 11 columns (121 elements). Element ARRAY(10,4) has a value of 7.123.

Arrays with dimensions that contain more than 10 or 11 elements must be explicitly dimensioned by a DIM or COMMON statement. Thus, without the appropriate DIM or COMMON statement, the following statements would both cause errors:

```
A(15) = 22.4
B(3,20) = 66.6
```

REDIMENSIONING

Once an array has been dimensioned, either explicitly by a DIM or COMMON statement, or implicitly through usage, it cannot be explicitly dimensioned again, but it can be redimensioned.

Arrays may be dynamically redimensioned by MAT assignment statements and MAT references in input lists within I/O statements. The number of dimensions and the extents of those dimensions may be changed as long as the number of elements in the original array is not exceeded.

Redimensioning only changes the view of the storage associated with an array, not the contents of the storage. Redimensioning may

cause an array to become smaller such that excess elements are not accessible, but the values remain in storage so that a subsequent redimensioning may bring them back into view.

Example

Original Dimensions	Redimensioning Reference
OPTION BASE 1	
ARRAY1(100)	MAT ARRAY1 = ARRAY1(85)
ARRAY1(100)	MAT ARRAY1 = ARRAY1(10,10)
ARRAY2(20,20)	MAT ARRAY2 = ARRAY2(300)
ARRAY2(20,20)	MAT ARRAY2 = ARRAY2(500) (See Note)

Note: This redimensioning reference is invalid; the redimensioned ARRAY2 would exceed the original size of ARRAY2.

Redimensioning COMMON Arrays

Arrays in COMMON can be redimensioned and the effects are global (remain in effect) across program units. If a subprogram redimensions a COMMON array, the new dimensions remain in effect when the subprogram is exited.

Redimensioning Parameters

An array that is a parameter (that is, appears in a SUB statement) may be redimensioned within a subprogram. When control returns to the calling program, the array retains its changed dimensions.

Example

Calling Program	Called Program
	100 SUB SUBPROG (ARRAY2(,))
100 OPTION BASE 1	110 OPTION BASE 1
110 DIM ARRAY1(10,10)	120 MAT ARRAY2 = ARRAY2(5,5,4)
120 CALL SUBPROG(ARRAY1(,))	
.	.
.	.
.	.

After control returns to the main program, ARRAY1 has the dimensioning ARRAY1(10,10).

EXPRESSIONS

Expressions are representations of numeric or character values, for example, variable or constants appearing above or in combination with operators.

An expression can be any one of the following:

Numeric Numeric values, optionally combined by numeric operators.

Character Character values, optionally combined by character string operators.

Relational Combinations of numeric expressions combined by relational operators or character expressions combined by relational operators.

Logical Combinations of relational expressions combined by logical operators.

Array Entire numeric or character arrays, optionally combined by numeric or character operators.

NUMERIC EXPRESSIONS

A numeric expression can be a numeric constant, a simple numeric variable, a reference to an element of a numeric array, a numeric-valued function reference, or a sequence of the above appropriately separated by numeric operators and parentheses.

There are five numeric operators, sometimes referred to as scalar operators, as shown in Figure 5.

EVALUATION OF NUMERIC EXPRESSIONS

IBM BASIC evaluates numeric expressions from left to right, subject to the evaluation order of the various operators defining the order of execution, as shown in Figure 5.

Operator	Meaning	Evaluation Order
** or ¬ or ^	exponentiation	1
*	multiplication	2
/	division	2
+	addition (or sign operator)	3
-	subtraction (or sign operator)	3

Figure 5. Numeric Operators and Evaluation Order

The double asterisk, the logical not sign, or the circumflex can be used for exponentiation; the one used depends upon the characters available on the terminal.

The left-to-right processing of operators is modified to provide compatibility with accepted practices of arithmetic processing. The highest processing priority is provided to exponentiation. The intermediate processing priority is provided to multiplication and division. If these two operators are encountered in the same expression, normal left-to-right

processing priority is used. The lowest processing priority is assigned to addition and subtraction. If these two operators are encountered in the same expression, normal left-to-right processing priority is used. See Figure 5 on page 25.

Evaluation of this expression:

Example

```
5+15-3
```

results in the value of 17: 5 is first added to 15; then 3 is subtracted from that sum.

In the following expression, the normal left-to-right process is overridden by the priority of operators:

Example

```
5+15/3
```

First, 15 is divided by 3; then, the quotient 5 is added to 5. The result is 10.

If an arithmetic statement contains several operations with mixed priorities, the operations with the higher priority are processed first in a left-to-right sequence. When that is completed, the next lower priority level of operations is processed.

Example

```
4+10/2-6**3/4+5
```

This expression is evaluated as follows:

1. Exponentiation is the first priority, so 6**3 is evaluated first, giving the following intermediate expression:

   ```
   4+10/2-216/4+5
   ```

2. Division is the next priority, evaluated in left-to-right order, so 10/2 is next evaluated, giving the following intermediate expression:

   ```
   4+5-216/4+5
   ```

3. Next, the right-hand division operation, 216/4, is performed, giving the following intermediate expression:

   ```
   4+5-54+5
   ```

4. Addition and subtraction are of the same priority; they are processed from left to right, giving the final result, which is -40.

PARENTHESES IN NUMERIC EXPRESSIONS

Parentheses provide a means to modify any of the above stated rules. The evaluation of parenthetic expressions has the highest possible priority. If more than one parenthesized subexpression is contained within an expression, the left-to-right priority becomes effective.

When parentheses are nested, the innermost pair of parentheses (the pair deepest nested) has the highest priority.

Using the expression from the previous example, we will change the order of processing by adding parentheses:

Example

```
4+10/2-6**3/(4+5)
```

1. Now the first evaluation is (4+5), giving the following intermediate expression:

   ```
   4+10/2-6**3/9
   ```

2. Exponentiation, 6**3, now takes place, giving the following intermediate expression:

   ```
   4+10/2-216/9
   ```

3. The two division operations, 10/2 and 216/9, now take place in left-to-right order, giving the following intermediate expression:

   ```
   4+5-24
   ```

4. Last of all, addition and subtraction generate the final value of -15.

Operators may not be presented in succession. The expression

```
4+-10
```

is invalid, whereas 4+(-10) is valid.

ADDITION AND MULTIPLICATION RULES IN NUMERIC EXPRESSIONS

In IBM BASIC, multiplication and addition are both commutative; in other words, A*B is the same as B*A and A+B is the same as B+A.

However, multiplication and addition are not always associative; that is, A*(B*C) does not necessarily give the same results as (A*B)*C. This is due to the situation shown below, where an overflow or underflow could result.

For example:

```
1E60*(1E20*1E-20) equals 1E60
```

but

```
(1E60*1E20)*1E-20 results in an overflow.
```

The expression in the parentheses causes the overflow and, if no ON OFLOW GOTO statement has been previously executed, causes the value 0.99999999999999999E+75 (BASIC infinity) to be substituted into the expression that is then multiplied by 1E-20. This results in the total expression equaling 0.99999999999999999E+55.

A/B is defined as A divided by B. If B=0, a division by zero (ZDIV) error will occur.

A-B is defined as A minus B. No special conditions exist.

PLUS AND MINUS AS SIGN OPERATORS

The + and - signs can also be used as positive/negative operators, which can be used in only two situations:

- Following a left parenthesis and preceding a numeric expression.
- As the leftmost character in an entire numeric expression.

Example

Valid	Invalid
-A+B	-A++B
-A+(-B)	-A+-B
B-(-2)	B--2

MIXED TYPE NUMERIC EXPRESSIONS

The result of an expression containing any decimal operands is decimal.

The result of an expression containing only integers is integer, except for division and exponentiation, where integer operands are converted to decimal, and the result is decimal.

If an integer operand is combined with a decimal operand, the integer operand is converted to decimal and the result is decimal.

Decimal or integer results may be assigned to numeric variables and arrays of either type (see "Assignment Statements" on page 68).

The IFIX, INT, and DEC intrinsic functions are useful with mixed numeric expressions. The IFIX function returns the rounded integer value of the argument. The INT function returns the largest integer not greater than the argument. The DEC function converts the argument to internal decimal format (see "Function Descriptions" on page 36).

CHARACTER EXPRESSIONS

Character expressions are made up of combinations of character constants, character variables, character array elements, and references to character functions, combined by the concatenation operator and modified by substring qualifiers.

Example

```
"ABCDEFG123456"  (character constant)
ALPHA$ & BETA$   (concatenation of 2 character variables)
"SER" & "IAL"    (concatenation of 2 character constants)
ZEBRA$(2:6)      (substring of a character variable)
CHR$(CHAR)       (a character array element)
GAMMA$(I,I)(4:9) (substring of a character element)
```

CONCATENATION

Concatenation is joining two character expressions with an ampersand (&), the concatenation operator. When two or more character strings are concatenated, the length of the resulting string is the sum of the individual string lengths.

Example

```
110 A$ = "MINNE"
120 B$ = A$ & "SOTA"
```

In this example, the character variable A$ is concatenated with the character constant "SOTA" to form the value of B$ (MINNESOTA). In the preceding example, A$ has a length of five ("MINNE"), "SOTA" has a length of four, and B$ has a length of 5+4=9 (with the value "MINNESOTA").

SUBSTRINGS OF CHARACTER VARIABLES AND ARRAYS

A character substring is a contiguous portion of a character string. A substring is identified by a substring qualifier. Operations to extract, insert, replace and append substrings are provided by references using substrings or qualifiers.

A substring of a character string is specified by adding a substring qualifier to a character variable or character array element. A substring qualifier has the form:

```
(m:n)
```

Where:

m is the beginning position of a string

n is the ending position.

Both m and n may be numeric expressions. When m and n are evaluated, the values used are the rounded integer equivalents of the expressions.

If m is less than 1, m is considered to be 1. If m is greater than the number of characters in the value associated with A$, the addressed substring is the null string immediately following the last character of A$. The number of characters in the value associated with A$ can be expressed as the intrinsic string function LEN(A$). If n is greater than LEN(A$), n is considered to be equal to LEN(A$). If m is greater than n, the addressed substring is the null string preceding the mth character of A$.

When the string notation occurs to the right of the equal sign in an assignment statement, extraction is indicated. When the string notation occurs to the left of the equal sign, replacement or insertion is indicated.

Examples

Assume that A$ contains the value ABCDEF and that B$ has the value VWXYZ. Following are examples of substring extraction, replacement, and insertion.

Extraction

`G$ = B$(2:3)`	Assigns WX (the 2nd and 3rd positions of B$) to G$.
`G$ = B$(4:4)`	Assigns Y (the 4th position of B$) to G$.
`G$ = B$(0:2)`	Assigns VW (the first two positions of B$) to G$. (m is taken as 1).
`G$ = B$(7:8)`	Assigns a null string to G$ (because B$ has only 5 characters)
`G$ = B$(4:8)`	Assigns YZ to G$. (n is taken as 5, which is the length of the string).

Replacement

`A$ (3:4) = "PQ"`	Causes CD to be replaced in A$ by PQ. ABPQEF
`A$ (3:4) = ""`	Causes CD to be deleted from A$. ABEF
`A$ (3:4) = B$(1:2)`	Causes CD to be replaced by VW. ABVWEF
`A$ (3:4) = B$(3:3)`	Causes CD to be replaced by X. ABXEF
`A$ (3:4) = B$(1:4)`	Causes CD to be replaced by VWXY. ABVWXYEF
`A$ (3:2) = B$(1:4)`	Causes VWXY to be inserted before C, resulting in ABVWXYCDEF.

Preceding Insertion

```
A$(1:0) = "PQ"        Causes insertion before the current
                      characters of A$.
                      PQABCDEF
A$(1:0) = B$(3:4)     Results in XYABCDEF.
A$(1:0) = B$(1:1)     Results in VABCDEF.
```

Trailing Insertion

```
A$(7:8) = "PQ"        Causes insertion after the current
                      characters of A$.
                      ABCDEFPQ
A$(7:8) = B$(2:4)     Results in ABCDEFWXY.
A$(10:11) = "PQ"      Results in ABCDEFPQ.
```

Substrings of Character Arrays

Substringing can also be performed on character array elements. For example, if B$(3) equals ABCDEFG, B$(3)(2:3) equals BC.

Character expressions can combine substring and concatenation operations to rearrange character strings. For example, the value returned by the DATE$ intrinsic function (which returns the date in the form YY/MM/DD) can be rearranged as follows:

```
50  C$ = DATE$
100 D$ = C$(4:8)&C$(3:3)&C$(1:2)
120 PRINT D$
```

This would return the date in the form mm/dd/yy instead of yy/mm/dd.

RELATIONAL EXPRESSIONS

A relational expression compares the value of either two numeric expressions or two character expressions. The expressions to be compared are evaluated and then compared according to the definition of the relational operator specified. According to the result, the relational expression is either satisfied (true) or not satisfied (false).

Relational expressions can appear in a BASIC program only as part of IF, DO, LOOP, EXIT IF, and CASE (in abbreviated form) statements.

RELATIONAL OPERATORS

Relational operators are defined as shown in Figure 6.

Operator	Definition
= or EQ	equal
<> or >< or NE	not equal
>= or => or GE	greater than or equal
<= or =< or LE	less than or equal
> or GT	greater than
< or LT	less than

Figure 6. Relational Operators

NUMERIC DATA IN RELATIONAL EXPRESSIONS

If a numeric expression of type integer is compared to an expression of type decimal, the integer expression is converted to decimal before the comparison is made.

CHARACTER DATA IN RELATIONAL EXPRESSIONS

When character data appears in a relational expression, it is compared character-by-character, from left to right, according to the COLLATE entry on the OPTION statement.

When OPTION COLLATE NATIVE is specified, the EBCDIC collating sequence is used. When OPTION COLLATE STANDARD is specified, the ASCII collating sequence is used. Both collating sequences are listed in "Appendix B. Character Set Collating Sequences" on page 327.

Figure 7 shows examples of the differences between the two options, COLLATE NATIVE and COLLATE STANDARD, when evaluating character expressions.

Expression	Native Result	Standard Result
"ABC"="ABC"	True	True
"ABLE"<"BALL"	True	True
"123">"BALL"	True	False
"$12"<"7"	True	True
"?55"<"44"	True	False

Figure 7. COLLATE Option and Comparisons of Character Expressions

Character expressions of different lengths can never be equal. If two character expressions are equal for the length of the shorter expression, the shorter expression is less than (in value) the longer expression. For example, "ABC" is less than "ABC ".

LOGICAL EXPRESSIONS

Relational expressions can be combined using the AND, OR, and NOT operators to yield a logical expression.

Logical expressions can be used in IF, DO, EXIT IF, and LOOP statements.

AND LOGICAL OPERATOR

AND specifies that both of two expressions must be true in order for the logical expression to be true.

Example

```
A = 3 AND NAME$ = "CHARLIE"
```

This logical expression is true only if both the numeric variable A equals 3 and the character variable NAME$ equals CHARLIE.

OR LOGICAL OPERATOR

OR specifies that either of two expressions must be true in order for the relational expression to be true.

Example

```
A = 3 OR NAME$ = "CHARLIE"
```

This logical expression is true if A equals 3 and/or NAME$ equals CHARLIE.

NOT LOGICAL OPERATOR

NOT specifies the negation of a relational or logical expression that follows the NOT operator.

Example

```
NOT(A EQ B)
```

This expression is true if A is not equal to B.

Note: However, A NOT EQ B is invalid, because EQ is not a relational or logical expression.

COMBINING LOGICAL EXPRESSIONS

Logical operators may be combined to form more complex logical expressions.

The evaluation of parenthesized expressions has the highest possible priority. If more than one parenthesized expression is contained within an expression, the left-to-right priority becomes effective. When parentheses are nested, the innermost pair of parentheses has the highest priority.

Example

```
NOT (NUM<=COUNT OR DATA$ = "END") AND ALPHA$<>BETA$
```

is a valid logical expression. The parentheses are required to negate the result of the OR operation.

PRIORITY OF EXPRESSION EVALUATION

The priority of evaluation of expressions is shown in Figure 8.

Subexpression or Operator	Evaluation Order
Arithmetic or character expressions	1
Relational operators	2
NOT	3
AND	4
OR	5

Figure 8. Scalar Expressions—Evaluation Priority

ARRAY EXPRESSIONS

Array expressions perform operations on the entire collection of a numeric or character array's elements rather than on each element individually, as scalar expressions do.

Array expressions may appear only within the MAT statement. See "MAT (Array Assignment) Statement" on page 183 for a description of array expressions.

A function is a named expression or block of statements that computes a single value. A function can be invoked through a reference to it in an expression.

You can define and name your own functions by using the DEF statement. See the user-defined functions "DEF Statement" on page 109 and "FNEND Statement" on page 126.

An intrinsic function is a predefined function supplied by IBM BASIC to evaluate commonly used mathematical, character, and system operations.

When an intrinsic function name ends in a dollar sign (for example, DATE$) a character, rather than numeric, value is returned.

An intrinsic function may require an argument list (of one or more arguments) to follow the function name. The argument list is enclosed in parentheses, and each item in the list is separated from the next by commas. The type of argument allowed for each intrinsic function is predefined, and an invalid argument produces an error. A function reference may be used anywhere in an expression where a variable, a constant, or an array reference of the same data type as the function return value, can be used.

If the value of a parameter is outside the allowable range, an exception is generated at the time the function reference is attempted. Appendix A, "Exception Codes," includes all exceptions caused by intrinsic function parameter errors.

Notation Used for Parameters

In this section, those intrinsic functions which require parameters are indicated by a parenthetic list after the function name. The notation used for parameters is:

Notation	Meaning
X or Y	Numeric parameter which can be either decimal or integer Example: ABS(X)
M or N	Type integer parameter. The function can be used with any numeric expression as the argument, but if the result of the expression is type decimal, it will automatically be converted to integer (with rounding). Example: FILE(M)
A or B	Array parameter. Example: DET(A)
A$,B$,or C$	Character variable parameter. Example: LEN(A$)
[ ]	Intrinsic functions which may or may not have parameters are indicated by enclosing the parameter list in square brackets. Example: RND[(X)]

INTRINSIC NUMERIC FUNCTIONS

The following numeric functions are supported:

```
ABS(X)                  LINE
ACOS(X)                 LOG(X)
ANGLE(X,Y)              LOG2(X)
ASIN(X)                 LOG10(X)
ATN(X)
                        MAX(X,Y[,...])
CEIL(X)                 MIN(X,Y[,...])
CEN(X)                  MOD(X,Y)
CNT
CODE                    PI
COS(X)                  PRD(A)
COSH(X)
COT(X)                  RAD(X)
CSC(X)                  REC(M)
                        REM(X,Y)
DATE                    RLN(M)
DEC(X)                  RND[(X)]
DEG(X)                  ROUND[(X,N)]
DET[(A)]
DOT(A,B)                SEC(X)
                        SGN(X)
EPS                     SIN(X)
ERR                     SINH(X)
EXP(X)                  SIZE(A[,M]) or SIZE(A$[,M])
                        SQR(X)
FAH(X)                  SRCH(A,X[,Y])
FILE(N)                 SUM(A)
FILENUM
FP(X)                   TAN(X)
                        TANH(X)
IFIX(X)                 TIME
INF                     TRUNCATE(X,N)
INT(X)
IP(X)                   UDIM(A,M) or UDIM(A$,M)

KEYNUM
KLN(M)
KPS(M)
```

INTRINSIC STRING FUNCTIONS

The following string functions are supported.

```
CHR$(M)                 POS(A$,B$[,M])

DAT$<(M)>               RPAD$(A$,M)
DATE$                   RPT$(A$,M)
                        RTRM$(A$)
FILE$(M)
                        SREP$(A$,M,B$,C$)
JDY<(C$)>               STR$(X)
                        TIME$
LEN(A$)                 TIME$
LPAD$(A$,M)
LTRM$(A$)               UPRC$(A$)
LWRC$(A$)
                        VAL(A$)
ORD(A$)
```

FUNCTION DESCRIPTIONS

The intrinsic numeric and string functions, in alphabetic order, are described in detail below:

ABS(X)

Returns the absolute value of X. The argument type can be integer or decimal. The type of the result is the same as the type of the argument.

Example

```
ABS(-1.2) is 1.2; ABS(3.4) is 3.4
```

ACOS(X)

Returns the arccosine (the inverse function of the cosine) of X, where X is in radians. X can be integer or decimal type. X must be in the range -1 to 1. The result is decimal and the range is 0 to PI.

Example

```
ACOS(0) is 1.5707963267948966
```

ANGLE(X,Y)

Returns the angle in radians between the positive X-axis and the vector joining the origin to the point with coordinates (X,Y); the angle is in the range: -PI to PI. X and Y may be integer or decimal. The result is decimal.

Example

```
ANGLE (0,0) is 0
```

ASIN(X)

Returns the arcsine (the inverse function of the sine) of X, where X is in radians. X must be in the range -1 to 1 and may be decimal or integer. The result type is decimal and in the range -PI/2 to PI/2.

Example

```
ASIN(1) is 1.5707963267948966
```

ATN(X)

Returns the arctangent (the inverse function of the tangent) of X, where X is in radians. X may be integer or decimal. The result is type decimal in the range -PI/2 to PI/2.

Example

```
ATN(1) is 0.78539816339744829
```

CEIL(X)

Returns the smallest integer greater than or equal to X (the ceiling function). Both its argument and function types are numeric (integer or decimal). The result type is the same as the argument type.

Example

```
CEIL (-1.2) is -1; CEIL (2.3) is 3
```

CEN(X)

Returns the degrees Centigrade corresponding to X degrees Fahrenheit. ((X-32)*5/9). X can have integer or decimal type and must be greater than or equal to -459.67. The result type is decimal.

Example

```
CEN(32) is 0
```

CHR$(M)

Returns the character corresponding to a specified position within the current collating sequence. Its argument is numeric (numeric values are rounded to integers if necessary); its result type is character. The result depends on the current setting of OPTION COLLATE (see "OPTION Statement" on page 211).

Example (ASCII)

```
CHR$(53) is "5",CHR$(65) is "A"
```

Example (EBCDIC)

```
CHR$(194) is "B",CHR$(241) is "1"
```

CNT

Returns the number of data items successfully processed by the last I/O statement executed. If input data was terminated with the solidus or slash (/) to indicate the end of data, CNT reflects the number of data items processed prior to the solidus character.

Example

```
* 100 DIM B(2): OPTION BASE 0
* 110 PRINT "ABCDE",12345,MAT B
* 120 A=CNT
* 130 PRINT A
```

As a result of the PRINT statement at line 110, the number of data items processed (5) is stored in A and printed at line 130.

CODE

Returns the value forwarded by the host system, when that system detected an error. The value of CODE is system dependent. See "Exception Handling Statements" on page 84.

COS(X)

Returns the cosine of X, where X is in radians and ABS(X) < PI*(2**50). The argument type may be integer or decimal. The result type is decimal.

Example

```
COS(PI) is -1
```

COSH(X)

Returns the hyperbolic cosine of X. X can be integer or decimal and ABS(X) < 175.366. The result is decimal.

Example

```
COSH(0) is 1
```

COT(X)

Returns the cotangent of X, where X is in radians and ABS(X) < PI*(2**50). The argument type may be integer or decimal. The result type is decimal.

Example

```
COT(PI/4) is 1
```

CSC(X)

Returns the cosecant of X, where X is in radians and ABS(X) <PI*(2**50). The argument type may be integer or decimal. The result type is decimal.

Example

```
CSC(PI/2) is 1
```

DAT$[(M)]

Returns the Gregorian date in the format YYYY/MM/DD. Its argument is the Julian date and can be either integer or decimal type (decimal values are rounded). Its result type is character. If the argument is omitted, DAT$ returns today's date.

DAT$(M) returns the Gregorian date in the range 0000/03/01 to 9999/12/31 corresponding to Julian dates (values of M) in the range 1721120 to 5373484.

Example

```
DAT$(2369916) is "1776/07/04"
```

For systems with no date, the value returned is 0000/00/00.

DATE

Returns the current date in the decimal form YYDDD, where YY are the last two digits of the year and DDD is the number of days elapsed in the year. If there is no calendar available, the value returned by DATE is -1. The result type is integer.

Example

```
N% is DATE
```

If current date is May 9, 1977, N% is set to 77129.
If there is no calendar available, N% is set to -1.

DATE$

Returns the current date in the string representation YY/MM/DD. If there is no calendar available, the value of DATE$ will be 00/00/00. The function type is character; it has no argument.

Example

A$ is DATE$

If the current date is May 9, 1977, A$ is set to 77/05/09. If there is no calendar available, A$ is set to the string value "00/00/00".

DEC(X)

Converts X to an internal decimal format. The function type is decimal and the argument type numeric (integer or decimal).

Example

The function can be used to convert an argument to decimal before calling a function:

```
* 110 I% = 3
* 120 CALL DSUB (DEC(I%))
* 130 END
* 140 SUB DSUB(B)
* 150 PRINT B
* 160 END SUB
```

In the above example, the subprogram requires a decimal argument. In order to pass the value of I%, the value of I% must be converted to decimal.

DEG(X)

Returns the number of degrees of X, where X is in radians. X can be integer or decimal. The result is decimal.

Example

DEG(1) is 57.295779513082321

DET[(A)]

Returns the value of the determinant of the square numeric array A. The function type is decimal; the argument type is numeric. If the argument is omitted, DET returns the determinant of the last array inverted using the INV function in a MAT statement. If the argument is not a square matrix, an exception occurs.

DOT(A,B)

Returns the dot product of the vector A and the vector B. A and B must be one-dimensional arrays with the same number of elements. The arguments may be integer or decimal. The result is integer if both arguments are integer; otherwise, the result is decimal.

Example

```
* 100 DIM A(3),B(3)
* 200 DATA 3,4,5,-2
* 300 DATA -3,3,1,-3
* 400 MAT READ A,B
* 500 C = DOT (A,B)
* 600 PRINT C
* 700 END
* RUN
 14
```

EPS

Returns the smallest positive decimal number that the implementation allows.

Example

```
EPS is .1E-75
```

ERR

Returns the exception code of the last exception which occurred. Exception codes are listed in "Appendix A. Exception Codes" on page 319. ERR returns an integer type value. See "Exception Handling Statements" on page 84.

Example

```
* 110 ON ERROR GOTO 140
* 120 A(11) = 1
* 130 STOP
* 140 PRINT ERR;' SUBSCRIPT OUT OF BOUNDS'
* 150 END
* RUN
2001  SUBSCRIPT OUT OF BOUNDS
```

EXP(X)

Returns the exponential value of X, that is, the value of the base of natural logarithms (e=2.718281828459045l) raised to the power X. If EXP(X) is less than machine infinitesimal (.1E-75), its value is replaced by zero. X must be less than or equal to 174.673 and can be integer or decimal. The result is decimal.

Example

```
* PRINT EXP(1)
2.71828
```

FAH(X)

Returns the degrees Fahrenheit corresponding to X degrees Centigrade. X can be integer or decimal, and must be greater than or equal to -273.15. The result is decimal.

Example

```
FAH(0) is 32
```

FILE(N)

Returns a numeric value indicating the status of the file specified by N. (The value is modified by each access to the file, to contain the current file status). The function type is integer; the argument type numeric (integer or decimal). If the file is not open, a value of -1 is returned.

Numeric Value	Status
0	Operation specified occurred successfully.
1	File opened with default INPUT.
2	File opened with default OUTPUT.
3	File opened with default OUTIN.
10	End-of-file exception during input operation.
11	End-of-file exception during output operation.

Numeric Value	Status
20	Transmission error during input operation.
21	Transmission error during output operation.

FILENUM

Returns the file number (0 to 255) of the file in which an error has occurred. Additional data about the cause of the error can be obtained through the use of the function FILE(N). The name of the file can be obtained through the use of the function FILE$(M).

If no error has occurred, the value returned is zero (0).

Example

```
* 100 OPEN #1: 'ABC',INPUT,SEQUENTIAL,INTERNAL
.
.
.
* 300 READ #1: A$,B$ EOF 400
.
.
.
* 400 LET XYZ = FILENUM
* 410
```

XYZ is set to file number 1, when the READ statement transfers control to 400 on end-of-file.

FILE$(M)

Returns the filename associated with the file number M. The function type is character. The argument type is integer or decimal. Returns a null string if the file is not open.

Example

```
* 100 OPEN #1: 'ABC', INPUT,SEQUENTIAL,INTERNAL
.
.
.
* 400 A$ = FILE$(1)
```

A$ is set to 'ABC', the name of file #1.

FP(X)

Returns the sign of X times the fractional part of the absolute value of X. Both the function and argument are of type numeric (integer or decimal). The result type is the same as the type of the argument.

Example

FP(-1.2) is -.2; FP(3.4) is .4

IFIX(X)

Returns the rounded integer value of X. The function is integer type while the argument type can be decimal or integer.

Example

```
IFIX (3.5) is 4
IFIX (3.2) is 3
IFIX (-3.5) is -4
IFIX (-3.2) is -3
```

INF

Returns the largest positive decimal number that implementation allows.

Example

```
INF is .99999999999999999E+75
```

INT(X)

Returns the largest integer less than or equal to X. The function and argument are both type numeric (both either integer or decimal).

Example

```
INT(1.3) is 1; INT(-1.3) is -2
```

IP(X)

Return the integer part of the absolute value of X times the sign of X. Both argument and function types are numeric (both either integer or decimal). The result type is the same as the type of the argument.

Example

```
IP(-1.2) is -1; IP(3.4) is 3
```

JDY[(C$)]

Returns the Julian date for the corresponding Gregorian date, expressed as 'YYYY/MM/DD'. If the argument is omitted the current date is assumed. For a system with no date, value 0 is returned when the argument is omitted. The function is integer; its argument character.

The argument for JDY (C$) must be the form 'YYYY/MM/DD' where MM must be between 1 and 12 inclusive and DD between 1 and 31 inclusive. If the argument has a DD of 31 or less but higher than possible for the month (for example, 1900/04/31), it is taken as equivalent to the appropriate date of the next month (1900/05/01).

The Julian date range is 1721120 to 5373484 corresponding to a Gregorian date range of 0000/03/01 to 9999/12/31.

Example

```
* 100 J% = JDY ("1960/01/01")

J% is set to 2436935.
```

KEYNUM

Returns the number of the PF key which caused the SKEY condition. (See "ON Condition Statement" on page 203.) If an SKEY exception has not occurred, zero is returned.

Example

```
* 100 ON SKEY GOTO 980
* 110 INPUT A$
.
.
.
* 980 XYZ=KEYNUM
* 990 ON XYZ GOSUB 1000,2000,3000
```

XYZ is set equal to the PF key which is pressed.

KLN(M)

Returns the length of the embedded key (stated in bytes) for the file M. The function type is integer; the argument type can be either decimal or integer. If the file is not keyed, or is not currently open, a value of -1 is returned.

Example

```
* A% is KLN(1)
```

A% is set equal to the number of bytes in the key associated with file #1.

KPS(M)

Returns the byte position for the start of the embedded key for the file M. The function type is integer; the argument type can be either decimal or integer. If the file is not keyed, or is not currently open, a value of -1 is returned.

Example

```
* A% is KPS(1)
```

A% is set equal to the starting byte position of the key in each record for file #1.

LEN(A$)

Returns the number of characters in the value associated with A$. The function type is integer and the argument type is character.

Example

```
* A$ = 'THIS'
* LN = LEN(A$)
```

LN is set equal to 4.

```
* B$ = ''
* LN = LEN(B$)
```

LN is set equal to 0.

LINE

Returns the line number of the most recent statement whose execution caused a transfer of control due to an exception. If no exceptions have occurred, zero is returned. See "Exception Handling Statements" on page 84.

Example

```
* 110 ON ERROR GOTO 140
* 120 A(11) = 1
* 130 STOP
* 140 PRINT "SUBSCRIPT OUT OF BOUNDS AT LINE "; LINE
* RUN
SUBSCRIPT OUT OF BOUNDS AT LINE  120
```

LOG(X)

Computes the natural logarithm of the positive number represented by X. X can be integer or decimal; the result is decimal.

Example

```
LOG(2) is 0.69314718055994533
```

LOG2(X)

Computes the base 2 logarithm of the positive number represented by X. X can be integer or decimal; the result is decimal.

Example

```
LOG2(2) is 1
```

LOG10(X)

Provides the common logarithm (base ten) of the positive number represented by X. X can be integer or decimal; the result is decimal.

Example

```
LOG10(2) is 0.3010299956639812
```

LPAD$(A$,M)

Returns the string of M characters produced by concatenating M minus LEN(A$) spaces to the front of the value A$. If M is not greater than LEN(A$), then A$ is returned. The function type is character; the argument types are character and numeric.

Example

```
* A$ is LPAD$("ABC",5)

There are two spaces inserted in front of 'ABC' and
A$ is set to "  ABC".
```

LTRM$(A$)

Returns the value of string A$ with all leading space characters removed. Its function type and argument type are both character.

Example

```
* B$ = LTRM$("  ABC")

The leading blanks are removed and B$ is set to "ABC".
```

LWRC$(A$)

Returns the string of characters resulting from the value associated with A$ by replacing each uppercase letter in the string by its lowercase version. Both the function type and argument type are character.

Example

```
* B$ = LWRC$("ABc")
```

B$ is set to "abc".

MAX(X,Y[,...])

Returns the larger of its numeric (decimal or integer) arguments. If any one of the numeric arguments is decimal, the numeric function value is also decimal.

Example

If A=2 and B=5, the statement;

```
* C=MAX(A,B)
```

Sets C equal to 5.

```
* C=MAX(1.2234,1.2214)
```

Sets C equal to 1.2234

MIN(X,Y[,...])

Returns the smaller of its numeric (decimal or integer) arguments. If any one of the numeric arguments is decimal, the numeric function value is also decimal.

Example

If A=2 and B=5, the statement;

```
* C=MIN(A,B)
```

Sets C equal to 2.

MOD(X,Y)

Returns X modulo Y, that is, an integer (whole number) in the range of 0 to Y minus 1, representing the relationship of Y to X, where Y is a modulus. Both function and argument types are numeric (integer or decimal). MOD(X,Y)=X-Y*INT (X/Y) if Y is nonzero. If Y is zero, MOD(X,Y)=X.

Example

MOD (11,5) is 1; MOD (68,44) is 24; MOD (301,5) is 1.

ORD(A$)

Returns the ordinal position of the character named by the string associated with A$ in the collating sequence of the declared character set, where the first member of the character set is in ordinal position zero. The acceptable values of A$ are the single character graphics of the character set and the two- and three-character mnemonics of that set. The acceptable values for both character sets are shown in "Appendix B. Character Set Collating Sequences" on page 327. The function type is integer; the argument type character.

Example

For the standard character set (OPTION COLLATE STANDARD).

```
ORD("BS")=8

ORD("A")=65

ORD("5")=53

ORD("SOH")=1
```

PI

Returns the decimal constant 3.1415926535897932, the ratio of the circumference of a circle to its diameter.

Example

```
* 100 R=10
* 110 AR=PI*R**2
* 120 PRINT AR
* RUN
314.159
```

AR equals the area of the circle whose radius equals R.

POS(A$,B$)

Returns the character position within the value associated with A$, of the first character of the first occurrence of the value associated with B$. If there is no such occurrence, POS(A$,B$) will be zero. POS(A$,"") will be one. The function type is integer; the argument type character.

Example

```
IF A$ contains " 123-4.56" AND
   B$ contains "-"


* X = POS(A$,B$)

X is set equal to 5


* X = POS("LEARN YOUR ABCS", "ABC")

X is set equal to 12


* X = POS("ABC","123")

X is set equal to zero, as 123 does not occur in the first
string.
```

POS(A$,B$,M)

Returns the character position, within the value associated with A$, of the first character of the first occurrence of the value associated with B$, starting with the Mth character of A$. If the defined string does not exist within the designated portion of A$, the value returned is zero. The function type is integer, the argument types character and numeric. POS(A$,"",M) is M.

Example

If A$ has the value "GRANDSTANDING", then;

```
* X = POS(A$,"AN",1)
```

X is set to 3

```
* X = POS(A$,"AN",4)
```

X is set to 8 the search started after GRA, at letter N

```
* X = POS (A$,"AN",9)
```

X is set to 0 as NDING does not contain the letters AN

PRD(A)

Returns the product of the elements of the array specified by A. Both the function and argument types are numeric (both integer or both decimal).

Example

```
*  10 OPTION BASE 1
*  20 DIM ARA(4)
*  30 DATA 4,3,10,5
*  40 MAT READ ARA
* 100 ARAPROD = PRD(ARA)
```

ARAPROD is set to 600 which is the product of 4*3*10*5.

RAD(X)

Computes the number of radians in X degrees.

Example

```
* X = RAD(23)
```

X is set to .401426

RAD(180) equals 3.1415926535897932

REC(M)

Returns the number of the last record processed in file M. Returns a zero if no records have been processed. Returns -1 if the file is closed or is not a relative file. The function type is integer; the argument type is numeric.

Example

```
X = REC(1)
```

X contains the number of the last record either read or written.

REM(X,Y)

Returns the remainder X-Y*IP(X/Y) if Y is nonzero, and returns X if Y is zero. Both the argument and function types are numeric (both either integer or decimal).

Example

```
REM(17,3) is 2
REM(6,0) is 6
REM(-17,5) is -2
REM(-17,-5) is -2
REM(16,4) is 0
REM(-6.74,4) is -2.74
```

RLN(M)

Returns the length of the last record referenced for file M. Zero is returned if no records have been processed. Returns -1 if the file is closed. The function type is integer; the argument type is numeric.

Example

```
* X = RLN(1)
```

X contains the number of bytes in the last record read from or written to in file #1.

RND[(X)]

Provides the next pseudorandom number in an implementation supplied sequence of pseudorandom numbers uniformly distributed in the range 0 LE RND LT 1.

If the argument X is included, RND also assigns the value of X to the seed value for the pseudorandom number generator. X must be in the range 0 LE X LT 1.

Example

```
* 100 LET N = INT(RND*1000+1)
```

Generates a random number in the range of 1 to 1000 and assigns it to the variable N. Each time the statement is executed, N may contain a different value.

ROUND(X,N)

Returns the value of X, rounded to N decimal digits (that is, INT(X*10**IFIX(N)+.5)/10**IFIX(N)). The result has the same type as the argument X. The arguments can be integer or decimal.

Example

```
* 100 X=15.73591
* 110 R=ROUND(X,2)
```

R contains 15.74

ROUND(123.456,-1.5) is 100

RPAD$(A$,M)

Returns the string of M characters produced by concatenating M minus LEN(A$) spaces to the end of the value of A$. If M is not greater than LEN(A$), A$ is returned. The function type is character; the argument types are character and numeric.

Example

```
* A$ = RPAD$("ABC",5)
```

A$ contains "ABC "

RPT$(A$,M)

Repeats the string A$, M number of times. The function type is character; the argument types are character and numeric.

Example

```
* A$ = RPT$("*",3)

A$ contains ***
```

RTRM$(A$)

Returns the value of string A$ with all trailing spaces removed. Both the function and argument types are character.

Example

```
* 100 A$="AB CD  "
* 200 B$=RTRM$(A$)

B$ contains "AB CD"
```

SEC(X)

Returns the secant of X, where X is in radians. X can be integer or decimal and ABS(X) must be less than PI*(2**50). The result type is decimal.

Example

```
SEC(PI) is -1
```

SGN(X)

Returns -1 if X<0, 0 if X=0, +1 if X>0. The function type is integer; the argument type is numeric (integer or decimal).

Example

```
SGN(-2) is -1

SGN(10) is 1

SGN(0) is zero.
```

SIN(X)

Returns the sine of X, where X is in radians. X can have integer or decimal type. The absolute value of X must be less than PI*(2**50). The result is decimal.

Example

```
SIN(3) is 0.14112000805986738

SIN(PI/2) is 1
```

SINH(X)

Returns the hyperbolic sine of the number X. X can have integer or decimal type. The absolute value of X must be less than 175.366. The result is decimal.

Example

```
SINH(0) is 0
```

SIZE(A) OR SIZE(A$)

Returns the number of elements in the array A. The function type is integer; the argument type may be numeric or character.

Example

```
* DIM A(4),B(4,3)
* N = SIZE(A)
* X = SIZE(B)
```

N contains 5 and X contains 20 if OPTION BASE 0 is in effect.

N contains 4 and X contains 12 if OPTION BASE 1 is in effect.

SIZE(A,M) OR SIZE(A$,M)

Returns the number of elements in the Mth dimension of array A. The function type is integer; the argument types are any type array, and numeric.

Example

```
* 10 DIM A(10), B(4,3)
* 20 N = SIZE(A,1)
* 30 X = SIZE(B,1)
* 40 Y = SIZE(B,2)
```

If OPTION BASE 0 is in effect, N contains 11, X contains 5 and Y contains 4.

If OPTION BASE 1 is in effect, N contains 10, X contains 4 and Y contains 3.

SQR(X)

Return the square root of X. X must be a positive number, it can be integer or decimal. The square root is returned with decimal type.

Example

```
* X = SQR(9)
* Y = SQR(58)
```

X contains 3 and Y contains 7.61577

SRCH(A,X[,Y]

Searches the one-dimensional array A for the value X, optionally beginning the search with the Yth element of A. The value returned is the subscript of the element of A which first matches X. If a match is not found, a value of -1 is returned.

The function type is integer. The array A must be one-dimensional and can be numeric or character. The type of X must agree with the type of A, numeric or character. If numeric, X is converted from integer to decimal or vice versa, if necessary, to match the type of the array. If necessary, Y is rounded to an integer value.

SREP$(A$,M,B$,C$)

Returns a string whose value is A$, where, starting at position M in string A$, the nonoverlapping occurrences of string B$ are located, and those occurrences are replaced with string C$. The function type is character; the argument types are character and numeric.

Example

```
* 100 A$="ABCDEFG"
* 120 B$="DE"
* 130 C$="123"
* 140 D$=SREP$(A$,2,B$,C$)

D$ contains "ABC123FG"
```

STR$(X)

Returns the string that would be generated by the PRINT statement as the external representation of the value associated with the numeric argument X. No leading or trailing spaces are included in this representation. This is the inverse of the VAL function. The function type is character; the argument type is numeric (either integer or decimal).

Example

```
STR$(139) is "139"
STR$(12E30) is "1.2E+31"
```

SUM(A)

Returns the sum of the elements of a one-dimensional numeric array. The function type is the same as the type of the array; integer or decimal.

Example

```
Assuming the elements of array ARX contain the values,
5,27,6,13.

* 100 SUMM=SUM(ARX)

SUMM contains 51.
```

TAN(X)

Computes the tangent of X, where X is stated in radians (integer or decimal type). The absolute value of X must be less than PI*(2**50). The functional type is decimal.

Example

```
TAN(PI/4) is 1
```

TANH(X)

Computes the hyperbolic tangent of the number X (integer or decimal). The function type is decimal.

Example

```
TANH(0) is 0
```

TIME

Returns the time elapsed since midnight, expressed in seconds. The function type is integer.

Example

```
If the current time is 11:15 A.M.

* X = TIME

X is set to 40500

If no clock is available, the value of X is set to -1.
```

TIME$

Returns the time of day in 24-hour notation, the eight character positions HH:MM:SS. The function type is character, with no argument.

Example

```
If the current time is 11:15 A.M.

* A$ = TIME$

A$ is set to 11:15:00

If there is no clock available, the value of A$ is set to
99:99:99.
```

TRUNCATE(X,N)

Returns the value of X truncated to N decimal places following the decimal point. The argument type is numeric integer or decimal; the type of the result is decimal.

Negative values of N cause the function to return the integer part of X with the N rightmost digits set to zero.

Example

```
Assume X has been set to the value of PI
(3.14159265358979).

* 100 X=TRUNCATE (X,4)

X is set to 3.1415

TRUNCATE (1234.56,-2)

returns 1200.
```

UDIM(A,M) OR UDIM(A$,M)

Returns the upper limit of dimension M of array A. The function type is integer; the argument types are any type for A and numeric for M.

Example

If array A is dimensioned A(23,10,6), the statement;

```
100 U=UDIM(A,1)
```

would cause U to equal 23.

```
150 U=UDIM(A,3)
```

would cause U to equal 6.

UPRC$(A$)

Returns the string of characters resulting from the value contained in A$ by replacing each lowercase letter in the string by its uppercase version. Both the function and argument types are character type.

Example

```
UPRC$("abC2")
```

returns

"ABC2"

VAL(A$)

Returns the value of the numeric constant contained in A$ if the string contained in A$ is a numeric constant. Leading and trailing spaces in the string are ignored. The string must be a valid numeric input form (see "Numeric Constants" on page 14). If it is not a valid numeric form, an exception is generated. The exception can be handled by the CONV condition in an ON condition statement. If the evaluation of the numeric constant would result in a value which causes an underflow or overflow, the usual action for numeric underflow or overflow occurs. See "ON Condition Statement" on page 203. The function type is decimal; the argument type character.

Example

```
VAL ("123.5") is 123.5
VAL ("MCMXVII") causes an exception.
VAL ("123.5XY") causes an exception.
```

IBM BASIC FILE CAPABILITIES

A file is a collection of data which is stored together. Files can be either "internal" (data stored within a program unit) or "external" (data stored on a medium, such as disk, external to all program units.)

This section deals exclusively with external files. External files allow interchange of data within a program unit as well as between program units, programs, systems, and languages.

RECORDS

The collection of data values which comprise a file can be arranged so that sets of values form logically related units; for example, a company payroll file would contain the name, address, job classification, salary, and other pertinent information for each employee. The term record is used to describe a discrete collection of data fields, such as the information needed to process an employee's payroll check.

Fixed record-length files are those whose record lengths are all the same, that is, each record in the file has the same length as every other record in that file.

If a payroll file is to contain data about the employee's dependents, a file with fixed length records allows a fixed number of positions whether the employee has one dependent or twenty.

Variable record-length files are those whose records do not necessarily have the same length, that is the record lengths can vary. The record length specified for a variable record-length file defines the maximum record length for that file.

In a payroll file with variable-length records, fields in an employee's record can be added for each dependent, up to the maximum record length specified for that file.

The record type (fixed or variable) of a file is declared in the RECORDS clause of the OPEN statement for a file.

FILE ATTRIBUTES

All files have attributes which describe how the data is organized, how the data is formatted, and how the data can be accessed. These three attributes, organization, format, and access, interact according to the rules specified in this section.

FILE ORGANIZATION

File organization is the file attribute which describes how data is arranged on a file. The way in which a file is organized determines how it can be accessed, that is, sequentially or directly (see "File Access Mode" on page 56).

The file organization can be: SEQUENTIAL, STREAM, RELATIVE, and KEYED. The file organization is specified in the OPEN statement.

Sequential Organization

A file with sequential organization consists of records which are ordered serially in the sequence in which they were written. The first record occupies the first position in the file; the last record occupies the last position in the file, regardless of the contents of the records.

The only way to access a file with sequential organization is serially, beginning with the first record.

Stream Organization

A file with stream organization is a file with sequential organization in which each record consists of a single value in internal format (see "File Format (Type)" on page 56).

Relative Organization

A file with relative organization consists of a sequence of "record slots" of the same fixed length, which are used to contain the records. A record slot may be empty (null) or may contain a record. Each slot has a unique record number associated with it, beginning with one and continuing to the maximum number of records that can be contained in the file. The record number is not necessarily contained within the record.

A relative file may be read either directly by reference to record numbers, or sequentially. When access is sequential, null records are bypassed on input.

A relative file must be written by reference to record numbers.

Keyed Organization

A file with keyed organization consists of records identified by keys. A key is a string of characters contained at a specific location within the record.

A keyed file may be accessed sequentially, that is, in the order in which the keys collate, or directly by reference to the keys. File positioning can be made by reference to a portion of the key. Keyed organization requires that VSAM be used. (See OS/VS Virtual Storage Access Method: Programmer's Guide.)

FILE FORMAT (TYPE)

The file attribute which describes the format of records in a file is known as file format or type. The TYPE clause of the OPEN statement permits specification of the file format.

There are three file formats: DISPLAY, INTERNAL, and NATIVE.

Display Format

DISPLAY file format means that each record is a sequence of characters in the same format as characters being displayed on a print output device.

If the "OPEN Statement" on page 206 specifies DEVICE PRINTER for an output file, a carriage control character is prefixed to each record. If the OPEN statement specifies DEVICE 3800, a carriage control character followed by a font control character are prefixed to each record.

Internal Format

INTERNAL file format indicates that each record is written as a sequence of numeric and string values. These values are written in internal binary format, each value preceded by a type byte (indicating integer, decimal, or character); consequently, files in this format cannot be edited.

Native Format

NATIVE file format indicates that the contents of each record are to be formatted by FORM statements.

Files with keyed and relative organizations require native file format.

FILE ACCESS MODE

The file attribute which determines the I/O operations allowed on a file is known as file access mode. File access is specified by the ACCESS clause of the OPEN statement.

The file access mode can be: INPUT, OUTPUT, and OUTIN.

INPUT Access Mode

INPUT file access mode specifies that only read operations are permitted on the file while the current OPEN statement is in effect. No records can be written, replaced, or deleted.

OUTPUT Access Mode

OUTPUT file access mode specifies that only write operations are permitted on the file while the current OPEN statement is in effect. No records can be retrieved from the file.

OUTIN Access Mode

OUTIN file access specifies that both read and write data transfer operations are permitted on the file while the current OPEN statement is in effect. This is the only file access mode which permits rewriting or deletion of records. Sequential files may be extended in this mode by adding more records, but they cannot be shortened.

COMBINATIONS OF FILE ORGANIZATION AND FORMAT

Not all combinations of organization and format are acceptable. The valid combinations are shown in Figure 9.

Combination Name	Organization	Format
Display	Sequential	Display
Stream	Stream	Internal
Internal	Sequential	Internal
Native Sequential	Sequential	Native
Relative	Relative	Native
Keyed	Keyed	Native

Figure 9. Valid Combinations of Organization and Format

Allowable Combinations for File Access

Figure 10 illustrates which file access modes are permitted with the valid type and organization combinations.

Type and Organization	Access INPUT	Access OUTPUT	Access OUTIN
Display	X	X	
Stream	X	X	
Internal	X	X	X
Native Sequential	X	X	X
Relative	X	X	X
Keyed	X	X	X

Figure 10. File Access Modes

Allowable Combinations for File Record Type

Figure 11 shows which record types are allowed with the valid type and organization combinations:

Type and Organization	FIXED Records	VARIABLE Records
Display	X	X
Stream		X
Internal	X	X
Native Sequential	X	X
Relative	X	
Keyed	X	X

Figure 11. Record Types Valid with Each File Organization

FILE STATEMENTS AND FILE ATTRIBUTES

Not all file input/output statements can be used with all kinds of files. Figure 12 lists which statements can be used with each of the six legal combinations of format and organization. Figure 12 also notes possible uses of the various combinations.

Allowable Combinations for File Record Type

Figure 11 shows which record types are allowed with the valid type and organization combinations:

Type and Organization	FIXED Records	VARIABLE Records
Display	X	X
Stream		X
Internal	X	X
Native Sequential	X	X
Relative	X	
Keyed	X	X

Figure 11. Record Types Valid with Each File Organization

FILE STATEMENTS AND FILE ATTRIBUTES

Not all file input/output statements can be used with all kinds of files. Figure 12 lists which statements can be used with each of the six legal combinations of format and organization. Figure 12 also notes possible uses of the various combinations.

Format	Organization	Statements	Use
NATIVE	KEYED	READ USING WRITE USING DELETE KEY WRITE USING REREAD USING RESET KEY RESET BEGIN SCRATCH	You would use this file type when a particular piece of data that you're using is always unique (for example, employee number) and you want to access individual records rapidly. You could use this file organization for Inventory of parts, Employee data, Bank account data, etc. The file must be a VSAM file.
NATIVE	RELATIVE	READ USING WRITE USING DELETE REC RESET REC RESET BEGIN REREAD USING REWRITE USING SCRATCH	You would use this file type when you know that the different records can be numbered. You might use this for error messages, or if you're keeping track of the occurrence of a particular number.
DISPLAY	SEQUENTIAL	INPUT LINE INPUT PRINT RESET BEGIN RESET END SCRATCH	You would use this file type when you wanted to save data that is in a printer format. You might create a file which could be spooled to a printer or output to a tape.
INTERNAL	STREAM	INPUT GET PUT WRITE RESET BEGIN RESET END SCRATCH	You would use this file type when each record has a single value. The length of each record may vary, as the value in the record is described in the record. You could use this as a text file in which each word is a separate field.
NATIVE	SEQUENTIAL	READ USING WRITE USING REWRITE USING REREAD USING RESET BEGIN RESET END SCRATCH	NATIVE or INTERNAL sequential files can be used the same way. You would use one of these file types when you are saving data in consecutive sequence and you only want to "report" or read the data from the beginning. For example, a transaction register that maintains everything that is entered in the exact order it was entered.
INTERNAL	SEQUENTIAL	INPUT READ WRITE RESET BEGIN RESET END SCRATCH	The only difference between a SEQUENTIAL, NATIVE and a SEQUENTIAL, INTERNAL file is the format of the records themselves.

Figure 12. File Format, Organization, Statements, and Use

IBM BASIC STATEMENTS

Statements are instructions to BASIC to perform a task or operation when the program is executed. They are either executable or nonexecutable:

- Executable statements cause a program action, such as value assignment or printing.
- Nonexecutable statements describe information needed by the program, but cause no program action.

All statements are processed in line number sequence, regardless of the order of entry, unless the sequence is altered by control statements, function references, subprogram calls, CHAIN statements, or exceptions.

Example

```
100 LET A=B
150 LET C=D
140 LET G=H
130 IF K=L THEN GO TO 160
.
.
.
160 LET M=N
```

Even though the line numbers are not presented in sequence, they will be processed in the correct order; 100, 130, 140 (unless K=L), 150, 160 ...

All of the statements are listed alphabetically and discussed individually in "Statement Descriptions" on page 88.

However, many statements fall into subcategories of similar statements, and many statements must be used in combination with other statements. These subcategories are:

- Declarative statements
- Control statements
- Assignment statements
- Input/Output statements
- Program segmentation statements
- Exception handling statements
- Debugging statements

These categories and sets of statements are discussed in the following sections.

DECLARATIVE STATEMENTS

Five statements perform declarative functions.

These statements do not cause an action to occur at the point in the program where they appear. Instead, they specify characteristics of the program in general; this influences the entire program unit (main program or subprogram) within which they appear.

Declarative statements may appear anywhere in a program unit, even subsequent to other statements which they influence.

COMMON The COMMON statement defines variables and arrays in a common region of storage where they may be shared by program units (main programs and subprograms). COMMON also explicitly defines dimensions of common arrays and declares the maximum length of common character variables and character array elements.

DECIMAL The DECIMAL statement explicitly defines which identifiers in the program unit are to be assigned decimal type.

DIM The DIM statement explicitly defines dimensions of arrays, declares the maximum length of a character variable, or declares the maximum length of each element of character arrays.

INTEGER The INTEGER statement explicitly defines which identifiers in a program unit are to be assigned integer type.

OPTION The OPTION statement specifies various options for a program unit during program compilation and/or execution. Options explicitly stated (by using the OPTION statement) override any appended to a RUN or COMPILE command.

CONTROL STATEMENTS

Control statements direct the flow of execution of a program. Most of the control statements can be used to transfer control from one location in a program to another, rather than executing it in a sequential manner.

Control statements are divided into the following logical groups: branch control, subroutine control, loop control, and decision structure control.

BRANCH CONTROL STATEMENTS

Branch statements transfer control to the specified line number or line label.

GOTO Branches unconditionally to the specified line number or line label.

ON exp GOTO Branches, conditionally, to one of the elements in the line number/line label list associated with this statement. The value of the expression (exp) determines to which element of the list the program branches.

SUBROUTINE CONTROL STATEMENTS

Subroutines provide a method of defining a group of statements that can be executed from various parts of the program without duplicating them each time. Control transfers to the subroutine and, after execution of these statements, returns to the statement immediately following the location from which the program branched.

GOSUB Transfers control unconditionally to the specified line number or line label and saves the return location for subsequent transfer back.

ON exp GOSUB Transfers control, conditionally, to one of the elements in the line number/line label list associated with this statement, and saves the return location for subsequent transfer back. The value of the expression (exp) determines to which element of the list the program will transfer.

RETURN Returns control to the first executable statement following the last GOSUB or ON exp GOSUB statement executed in this program unit.

LOOP CONTROL STATEMENTS

Loop control gives programs the capability of executing a single statement or a group of statements any number of times. Loop control statements are also referred to as loop blocks; there are two types: the DO/LOOP block and the FOR/NEXT block.

Any type of loop block may be nested completely within any other type of loop block, as shown in Figure 13. The effect of loop nesting is that each time the outer block is executed once, the inner block is executed until the exit condition is met.

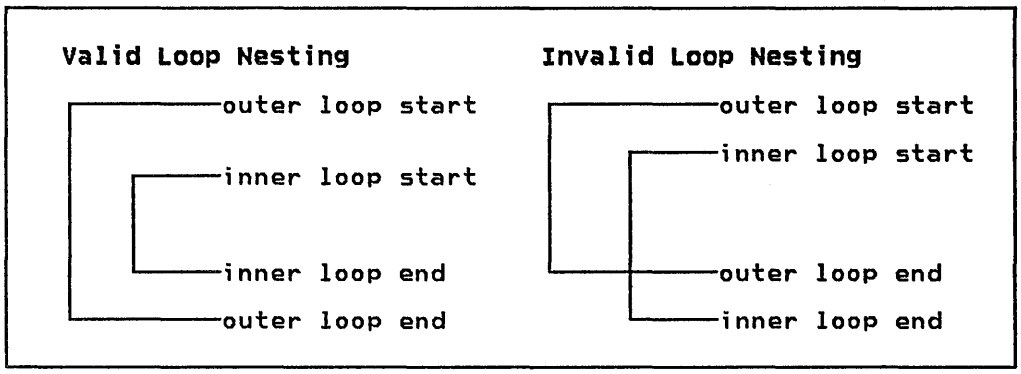


Figure 13. Valid and Invalid Loop Nesting

Loops can be nested to any depth needed by the logic of the program.

DO/LOOP Blocks

DO/LOOP blocks process a series of statements repeatedly WHILE or UNTIL one or more conditions are met.

DO The DO statement is the first statement (also referred to as the upper limit) of a DO loop. It can be used to control how long processing remains inside the loop by the use of the optional keywords WHILE or UNTIL:

- **WHILE** As long as a particular condition exists, the series of statements between the DO statement and the LOOP statement are processed repeatedly.
- **UNTIL** Until a particular condition is met, the series of statements between the DO statement and the LOOP statement are processed repeatedly.

The condition may be controlled by statements within the loop.

LOOP Indicates the last line (also referred to as the lower limit) of the DO loop.

The WHILE and UNTIL clauses can be used in the LOOP statement in the same manner as in the DO statement to control how long processing remains in the loop.

DO and LOOP statements must always be used in pairs with the DO appearing first in line number sequence.

Figure 14 shows the flow of control in a DO/LOOP block.

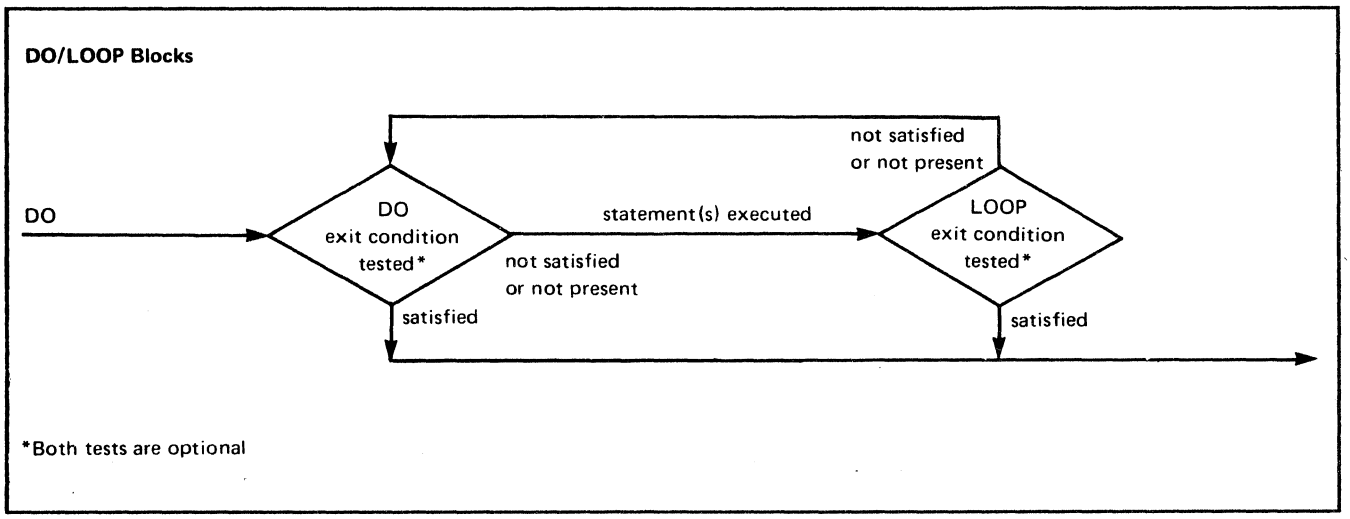


Figure 14. DO/LOOP Block Flow of Control

If the WHILE/UNTIL clauses are specified only in the DO statement, the exit condition is tested before the statements within the loop are executed. This means that, if the exit condition is true the first time the DO statement is encountered, the statements within the loop are not executed.

If the WHILE/UNTIL clauses are specified only in the LOOP statement, the exit condition is tested after the statements within the loop are executed. This means that, if the exit condition is true the first time the LOOP statement is encountered, the statements within the loop are executed once.

In one loop block, both the WHILE and UNTIL clauses can be specified both in the DO statement and in the LOOP statement, thus permitting a variety of exit conditions.

The only statements that may transfer control into the body of a DO loop are CONTINUE, RETRY, RETURN, and END SUB, each having been set originally by a condition within that DO loop.

A DO loop may be exited by an EXIT IF statement, as well as by other branching statements.

FOR/NEXT Blocks

FOR/NEXT blocks allow you to process a series of statements repeatedly until a count condition is met.

FOR The FOR statement is the first line of a FOR loop. It controls how long processing will remain in the loop by providing:

- A count condition control variable
- An initial value for the count condition control variable
- A final value for the count condition control variable
- An increment for testing the count condition control variable

The count condition control variable is incremented and tested after each iteration of the loop. When the count condition is met, loop processing stops.

NEXT The NEXT statement is the last line of a FOR loop and provides a lower limit to the loop.

Its causes the incrementing of the count condition control variable.

FOR and NEXT statements must appear in sets. The FOR must appear first in line number sequence.

The flow of control in a FOR/NEXT loop is shown in Figure 15.

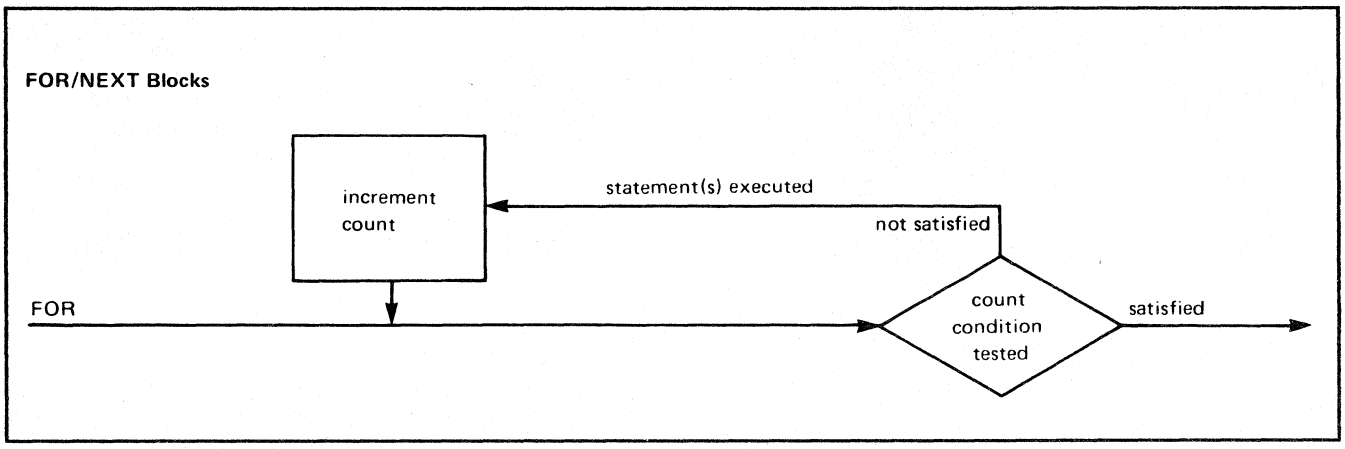


Figure 15. FOR/NEXT Loop Flow of Control

If the count condition is true the first time it is tested, the statements within the loop are not executed.

The only statements that may transfer control into the body of a FOR loop are CONTINUE, RETRY, RETURN, and END SUB, each having been set originally by a condition within that FOR loop.

A FOR loop may be exited by an EXIT IF statement as well as by other branching statements.

DECISION STRUCTURE CONTROL STATEMENTS

These statements are tools for structured programming. They allow conditional processing of alternative statement sequences. Block IF structures allow two alternative paths of execution. SELECT structures allow multiple alternative paths of execution.

Any decision structure can contain within it any other decision structure or loop.

IF Blocks

IF blocks provide for alternative paths of program execution, depending on a logical expression.

The path selected depends on whether the logical expression evaluates as true or false.

IF blocks contain four basic parts:

IF The keyword IF, followed by a logical expression to be tested.

THEN Block The keyword THEN, followed by a block of statements processed when the logical expression is true. After the block of statements is executed, control is transferred to the statement immediately following the END IF statement.

ELSE Block The keyword ELSE, followed by a block of statements processed when the logical expression is false.

If an ELSE block is not present, the statement immediately following the END IF is executed when the condition is false.

END IF The last line of the IF block.

Figure 16 shows this flow of control.

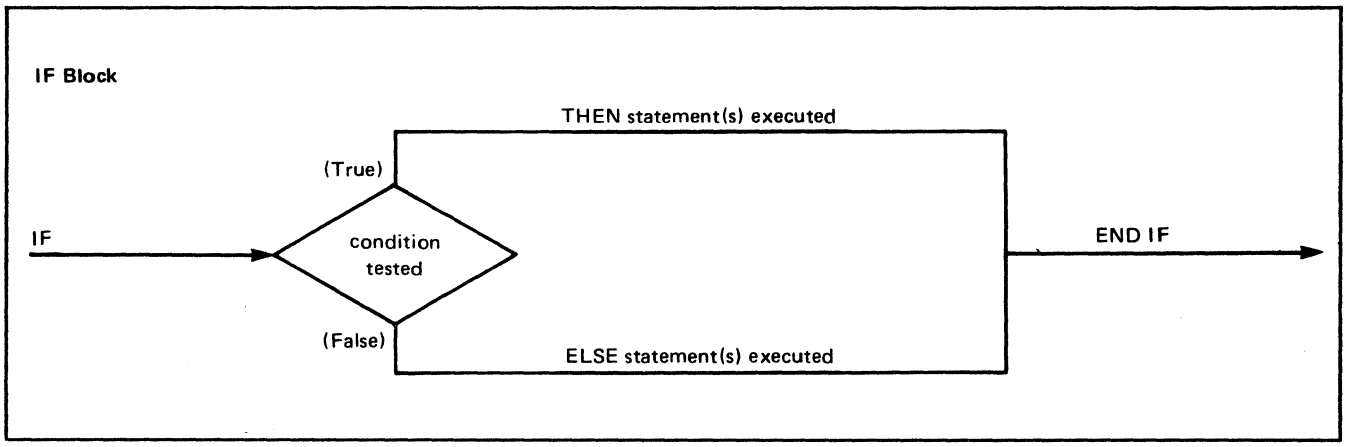


Figure 16. IF Blocks—Flow of Control

The following example shows how an IF block is coded:

Example

```
100 IF A=B THEN
110     LET C=10
120     LET E=20
130 ELSE
140     LET E=10
150 END IF
```

Note: The indentation clarifies the IF block structure.

The IF block in the example is executed as follows:

1. If A equals B, lines 110 and 120 are processed.
2. Processing then skips to the line after END IF.
3. If A does not equal B, lines 130 through 150 are processed.
4. Processing then continues at the line after END IF.

The only statements that may transfer control into an IF block (either the THEN or ELSE block) are CONTINUE, RETRY, RETURN, and END SUB, each having been set originally by a condition within that IF block. An IF block can be exited by branch control and exception handling statements.

IF STATEMENT: Another form of the IF statement allows for:

1. IF/THEN—conditional execution of a branch or list of imperative statements.
2. IF/THEN/ELSE—conditional execution of one of two branches or sequences of imperative statements.

This form of the IF statement, unlike the IF block, must be contained on a single line. In this form, if there is no ELSE

clause, the statement on the next line is executed when the condition is false.

SELECT Blocks

SELECT blocks conditionally process one of several alternative sequences of statements, depending on the value of a selection expression. (An IF block allows only two alternatives; the SELECT block allows many alternatives, based on the selection expression value.)

A SELECT block consists of four parts:

SELECT The SELECT statement is the initial delimiter of a SELECT block. It contains the select expression to be tested by the CASE statements that follow.

CASE The CASE statement is a selection statement in a SELECT block. The keyword CASE immediately precedes a case item, which is tested against the selection expression. If the case item tests true, the sequence of statements following the CASE statement is executed. There can be as many case items as are needed by the logic of the program.

The selection expression is evaluated and compared with the case items in the order in which the CASE statements occur until a match is found.

Once a match is found, that CASE block is executed.

After the appropriate CASE block, if any, has been processed, the program continues at the statement following the END SELECT statement (that is, the remaining CASE blocks are skipped).

CASE ELSE The CASE ELSE statement immediately precedes a sequence of statements executed if no case item tests true.

CASE ELSE is optional.

The CASE ELSE statement must be last block specified in the SELECT block.

If there is no CASE ELSE block and a match is not found, an exception occurs. This exception is of the ERROR category—see "Exception Handling Statements" on page 84 and "ON Condition Statement" on page 203.

END SELECT The END SELECT statement is the end delimiter of a SELECT block.

In line number sequence, it must be placed after the SELECT statement.

Figure 17 on page 67 shows the logical flow of control in a SELECT block.

clause, the statement on the next line is executed when the condition is false.

SELECT Blocks

SELECT blocks conditionally process one of several alternative sequences of statements, depending on the value of a selection expression. (An IF block allows only two alternatives; the SELECT block allows many alternatives, based on the selection expression value.)

A SELECT block consists of four parts:

SELECT The SELECT statement is the initial delimiter of a SELECT block. It contains the select expression to be tested by the CASE statements that follow.

CASE The CASE statement is a selection statement in a SELECT block. The keyword CASE immediately precedes a case item, which is tested against the selection expression. If the case item tests true, the sequence of statements following the CASE statement is executed. There can be as many case items as are needed by the logic of the program.

The selection expression is evaluated and compared with the case items in the order in which the CASE statements occur until a match is found.

Once a match is found, that CASE block is executed.

After the appropriate CASE block, if any, has been processed, the program continues at the statement following the END SELECT statement (that is, the remaining CASE blocks are skipped).

CASE ELSE The CASE ELSE statement immediately precedes a sequence of statements executed if no case item tests true.

CASE ELSE is optional.

The CASE ELSE statement must be last block specified in the SELECT block.

If there is no CASE ELSE block and a match is not found, an exception occurs. This exception is of the ERROR category—see "Exception Handling Statements" on page 84 and "ON Condition Statement" on page 203.

END SELECT The END SELECT statement is the end delimiter of a SELECT block.

In line number sequence, it must be placed after the SELECT statement.

Figure 17 on page 67 shows the logical flow of control in a SELECT block.

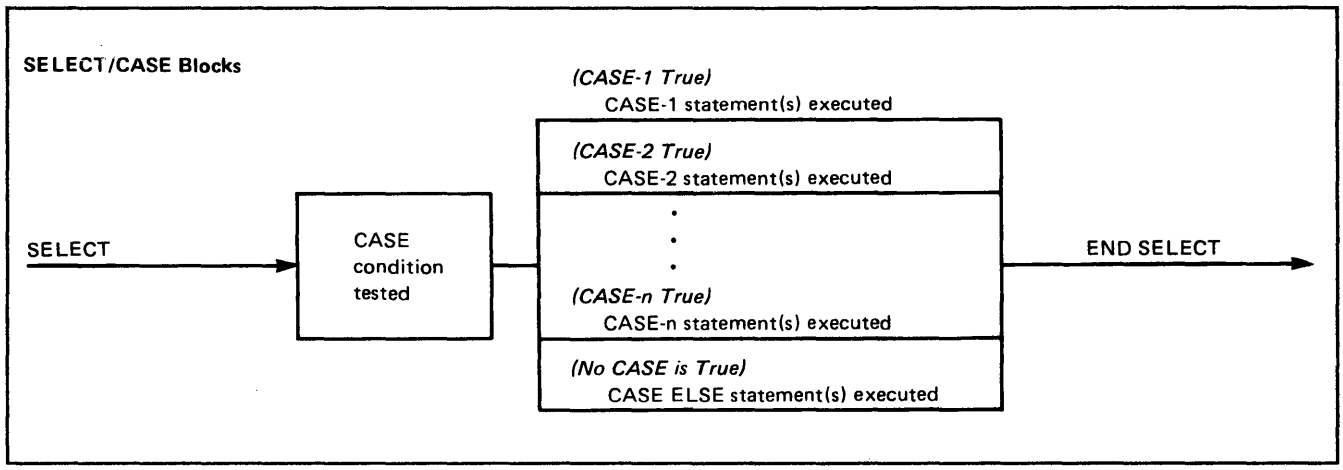


Figure 17. SELECT Block—Flow of Control

In the following SELECT block example, assume the variable ABC has a value of 5:

Example

```
100 SELECT ABC
110 CASE 6 TO 10      !SELECTED IF ABC HAS A VALUE 6 THRU 10
120     LET A=B
.
.
.
150 CASE LT 4         !SELECTED IF ABC HAS A VALUE LESS THAN 4
160     LET C=D
.
.
.
200 CASE 12           !SELECTED IF ABC HAS THE VALUE 12
210     LET E=F
.
.
.
250 CASE ELSE         !SELECTED IF NO OTHER CASE IS TRUE
260     LET G=H
.
.
.
300 END SELECT
```

This SELECT block is executed as follows:

1. Line 110 is evaluated and it is determined that ABC is not between 6 and 10, so processing skips to line 150.
2. Line 150 is evaluated and it is determined that ABC is not less than 4, so processing skips to line 200.
3. Line 200 is evaluated and it is determined that ABC is not equal to 12, so processing skips to line 250.
4. Line 250 is used to indicate processing in the event that none of the conditions specified by a CASE statement are met. In this example (ABC=5) these lines are processed.

There may be any number of CASE blocks within a SELECT block, but they must never overlap. Any illegal nesting of CASE and CASE ELSE blocks is detected either during compilation or as part of the RUN command.

A CASE block can be exited by branch control and exception handling statements.

Control may transfer into the body of a CASE or CASE ELSE block only through the use of a CONTINUE, RETRY, RETURN, or END SUB statement.

Further, you should ensure that all CASE blocks can be reached. For example, if the third CASE statement in the above example had a value of 2 (instead of 12), it could never be reached, because the second CASE block selects any value that is less than 4.

EXECUTION CONTROL STATEMENTS

Execution control statements provide a method of halting your program, temporarily stopping your program, or selecting a new starting point for a list of pseudorandom numbers.

END Indicates the end of a main program and therefore is both logically and physically the last statement in a main program. It halts the execution of your program and closes all files that have been opened.

PAUSE Temporarily halts the execution of your program, displays either a system message, or the message associated with the PAUSE statement itself, and resumes execution at the next statement when the GO command or a null entry are entered.

RANDOMIZE Is used to start a new series of pseudorandom numbers.

STOP Halts the execution of your program, and performs the same operations as the END statement. The STOP statement, however, unlike the END statement, may be placed anywhere in your program, and you may specify it more than once.

ASSIGNMENT STATEMENTS

Assignment statements assign (move) data to variables.

The data assigned can be a constant, another variable, the result of a function or an expression. The data and the variable must both be of the same type; that is, the variable must be a character variable if the data is character data, or the variable must be a numeric variable if the data is numeric.

For numeric data types (integer or dacimal), however, the data is always converted to match the numeric variable before it is assigned (moved). That is, an integer value is converted to a decimal value if the variable is decimal, or a decimal value is converted (with rounding) to an integer value if the variable is an integer. Numeric overflow may occur if decimal values are converted to integer.

The assignment statements are:

LET assigns values to scalar variables.

The keyword LET is optional.

MAT assigns values to arrays.

Some examples of assigning constants to variables are shown in Figure 18 on page 69.

LET A# = 123.56	Assigns decimal constant 123.56 to decimal variable A#.
LET B,C,D = 5.7	Assigns decimal constant 5.7 to variables B, C, and D.
LET E% = 3	Assigns integer constant 3 to integer variable E%.
LET F% = 3.666	Rounds decimal constant 3.666 to the value 4 and assigns it after rounding to integer variable F%.
LET ABC$ = "STRING"	Assigns the character string constant "STRING" to the character variable ABC$.
MAT ARAY# = (5)	Assigns integer constant 5 to every element in the decimal array ARAY#.
LET LIST = 5	Assigns integer constant 5 to variable LIST (a keyword is allowed as a variable name in an assignment statement only if the optional keyword LET is specified).

Figure 18. Assignment Statement—Assigning Constant Values

ROUNDING RULES

As Figure 2 on page 16 shows, the internal representation of a decimal value provides the capability of conveying 19 digits of decimal data; however, the actual representation provides for only 17. During the evaluation of a numeric expression, intermediate results make use of all 19 digits. At the completion of evaluation, the last two digits are examined and, if they represent a value equal to or greater than 50, the 17th digit is rounded up, and the 18th and 19th digits are set to zero.

Some examples of assigning one variable to another variable are shown in Figure 19 on page 70.

A# = B	Assigns the value in variable B to variable A.
B,C,D = X	Assigns the value in variable X to variables B, C, and D.
M$ = N$	Assigns the character string value in variable N$ to character variable M$. (In this case the maximum length of M$ must be greater than or equal to the length of the character string in N$ or string overflow occurs.
MAT AR = BRAY	Assigns the values in array BRAY to the array AR on an element-by-element basis. (When assigning values to arrays, be sure the array on the left of the equal sign has the proper dimensions. Any array can be redimensioned during an assignment, but its overall size cannot be increased).
DIM ABC$(5,5) MAT ABC$ = XYZ$(2,3)	Redimensions ABC$ to 2 by 3 and then assigns the values in array XYZ$ to the corresponding elements in array ABC$.

Figure 19. Assignment Statement—Assigning Variable Values

INPUT/OUTPUT STATEMENTS

Input/output statements define file attributes, define and control access to file data, and transmit file data to, from, and within a program.

Input/output statements are classified according to the disposition of the data being processed (internal or external), and according to the function being performed. The input/output statements are classified by function as follows:

- Internal Data Input/Output Statements
- Terminal Input/Output Statements
- File Input/Output Statements

After a description of general input/output considerations, the individual statements under each of these headings are described below.

GENERAL INPUT/OUTPUT CONSIDERATIONS

This section discusses input/output lists used by many statements, data rules for input/output, the use of FORM and IMAGE statements for data formatting, and input/output error processing.

Input/Output Lists

Most of the input/output statements that transmit data require an input/output list, a list of items associated with data to be transmitted either as input or output. The exception to this

requirement is the PRINT statement, which allows a null list. The null PRINT list produces a blank line (or record) of output.

The input/output list has the following format:

```
[ item separator   [item separator] ...] [item]
```

Where:

item
is either the name of a data item or a print clause, and may take these forms:

1. In an input list:
   a. Simple variable.
   b. Subscripted array element.
   c. Array, that is:

      MAT array name

   d. Array with redimensioning, that is:

      MAT array name

      (numeric expression [,numeric expression] ...).

      The rounded integer values of the expressions are used to redimension the array before storing input values into the array.
2. In an output list:
   a. Simple variable.
   b. Subscripted array element.
   c. Array, that is:

      MAT array name

   d. Scalar expression (either numeric or character).
   e. TAB clause (PRINT and PRINT file statement only)
   f. NEWPAGE clause (PRINT and PRINT file statement only)

separator
is one or more commas or semicolons, with the following restrictions:

- A semicolon is valid only on output, and only used when transmitting data to the terminal or to a DISPLAY file, that is, only in PRINT and PRINT file statements.
- The list may end with a comma or semicolon only in PRINT and PRINT file statements.
- More than one consecutive comma or semicolon for a separator is only allowed in PRINT and PRINT file statements.

If the list consists only of arrays, there is an alternative form of the statement, with MAT as the first word (that is, preceding the keyword that identifies the statement) and with no occurrence of MAT in the input/output list.

For example:

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

is equivalent to

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

Input/Output Data Rules

Unless stated otherwise, the following rules apply to data associated with input/output statements.

Input list Data received is assigned to the values in the list in the order received.

Numeric conversions to the type in the input list are performed.

A character variable in an input list assumes the length of the character data value.

The action when an error condition is detected varies according to the input statement used, whether the receiving data is from a file or from the terminal and the error processing that has been specified in the program. See the specific input statement for details.

Output list Data is moved in the order specified in the list. See the specific output statement for details.

FORM and IMAGE Statements

Many of the transmission input/output statements have a USING clause which refers to a FORM or an IMAGE statement to format data for output, or which refers to a FORM statement for input. This reference may be the line number or line label of a FORM or IMAGE statement, or a character expression which, when evaluated, is equivalent to a FORM or IMAGE statement.

Such a character expression is evaluated as follows:

- If FORM is specified, the FORM syntax is used.
- Otherwise, the IMAGE syntax is used.

FORM Character Expressions

When the FORM is entered as a separate statement, replication factors can be constants or variables, and the parameter n in the specifications X n, POS n, and SKIP n can be a numeric expression. When the FORM is contained in a character expression, however, these values must be constants.

In a FORM statement, the specifications X, POS, and SKIP must be followed by at least one space character. In FORM character expressions, the spaces are optional.

Input/Output Error Processing

During execution of input/output statements, various errors can occur which may be handled in a number of ways. The ON Condition statement can be used to provide general error-handling routines for many of the situations which might occur.

Most of the input/output statements permit you to specify error-recovery clauses for that particular statement, either as an EXIT clause or as individual error clauses, such as CONV,

SOFLOW, and IOERR. The EXIT clause, which refers to an EXIT statement, and the other error clauses are mutually exclusive. Multiple error clauses are separated by commas and cannot duplicate each other.

Error-recovery clauses on input/output statements temporarily override any ON Condition statements which are in effect for similar conditions.

INTERNAL DATA INPUT/OUTPUT STATEMENTS

An internal data file consists of a sequence of numeric and character values which exist within the program unit. These values can only be accessed sequentially and only for input.

The internal data input statements are:

DATA One or more DATA statements create an internal data table.

READ READ statements assign values from the data table to variables and arrays.

RESTORE The RESTORE statement resets the file pointer to the first value in the data table.

TERMINAL INPUT/OUTPUT STATEMENTS

Terminal input/output statements provide communication between a program and the terminal during an interactive session.

Terminal input/output statements are of two types: line-by-line input/output statements and screen field input/output statements.

Line-By-Line Input/Output Statements

This group of statements refer to the terminal one line at a time. The line-by-line statements are:

INPUT Provides data to a program from a terminal for assignment to the items in an input list.

LINE INPUT Provides unformatted data to a program from a terminal for assignment to a character item in an input list.

MARGIN Sets the boundaries for writing data via a PRINT statement to a terminal.

PRINT Writes both formatted and unformatted data from a program to a terminal. The output to the terminal is determined by the items in an output list.

Full Screen Input/Output Statements

Full screen input/output statements can be used with a display terminal that can read and write specific fields on the screen. The full screen input/output statements are:

INPUT FIELDS Reads one or more data values from one or more screen fields and assigns each to a variable.

PRINT FIELDS Displays one or more data values in one or more screen fields.

Attribute Characters on the Screen: On the IBM 3270 family of terminals, or equivalent, "attribute characters," which control screen attributes, occupy screen positions and display as blanks.

An attribute character precedes and follows each screen field accessed by an INPUT FIELDS or a PRINT FIELDS statement. These attribute characters are not available to the IBM BASIC user.

The screen positions occupied by these attribute characters are immediately to the left and right of the field the user specifies. The character to the "left" of a character in column one is the last character in the previous row, and the screen wraps around—that is, the bottom row "precedes" the top row. The same ordering (left to right, top to bottom with wraparound) determines the attribute character to the right of the field.

Overlapping Fields: Print fields may overlap other fields, but the visual characteristics of previous fields may change. For example, if the end of an output field overlaps the beginning of a previous high intensity field (H attribute), the portion of the previous field which is not overwritten is still displayed, but with normal intensity.

Input fields may not overlap existing attribute characters. Such an overlap causes an exception.

Mixed Mode Operations

Be careful when intermixing full screen operations with line-by-line operations. When switching between them, there is no implicit display screen clearing. Therefore, if the display screen is not cleared before full screen processing begins, full screen processing fields will be intermixed with the previous contents of the display screen.

To clear the screen, use the PRINT NEWPAGE statement.

FILE INPUT/OUTPUT STATEMENTS

File Input/Output statements process data which is stored in files, as described in the chapter "IBM BASIC File Capabilities" on page 54.

The file input/output statements are of two types: file control and file transmission. In addition, a number of file input/output statements have a file-positioning clause.

File Positioning Clauses

When a sequentially organized file is opened, it can be positioned at its beginning or its end; other file input/output statements directly or indirectly affect the position of the file. "Current position of the file" is equivalent to saying the record or value which would be accessed if a sequential input/output operation were to be executed at that time; the frequently used term "file pointer" can be viewed as an arrow pointing to the current position of the file.

When a relative or keyed file is opened, only the BEGIN option is available.

Several file input/output statements allow for a "positional" clause which specifies a particular position; this position is sometimes designated as the beginning or end of the file (RESET #fileref/RESTORE #fileref).

For relative files, the positional clause may specify a RECORD option; the file is positioned at the record whose relative position is identical to the one specified.

For keyed files, the positional clause may specify a KEY option; the file is positioned to the first record which satisfies the KEY option condition. The KEY option must specify a condition using the exact length of the key.

For keyed files, the positional clause may also specify a SEARCH option; the file is positioned to the first record which satisfies the SEARCH option condition. The SEARCH option condition can specify a partial key (that is, a key whose length is less than the length of the key).

The input/output statements which allow record positioning are shown in Figure 20.

File Statement	Options Allowed	
	Relative Files	Keyed Files
DELETE #fileref	RECORD	KEY
READ #fileref	RECORD	KEY SEARCH
RESET #fileref	RECORD	KEY SEARCH
RESTORE #fileref	RECORD	KEY SEARCH
REWRITE #fileref	RECORD	KEY
WRITE #fileref	RECORD	KEY

Figure 20. Positioning Options Allowed—File Input/Output Statements

File Control Statements

OPEN #fileref Activates a file, assigns a file reference number, specifies access, file type, organization, record type, and file position.

CLOSE #fileref Deactivates a file, preventing further access to it until that file is reactivated by another OPEN #fileref statement.

RESET #fileref Changes the position of the file pointer for a file.

RESTORE #fileref Is identical to the RESET #fileref statement.

MARGIN #fileref Specifies the page margins for display files which are written with PRINT file statements.

SCRATCH #fileref Erases the contents of a file and resets the file pointer to the beginning.

Each of the file input/output statements refers to a file by means of a file reference number (#fileref) which is assigned by the OPEN statement for that file; all other input/output statements for that file must refer to this number.

In most statements, the file reference number must be between 1 and 255; in some statements, it may be zero, which always specifies the terminal.

The file reference number assigned by an OPEN statement remains in effect, even across communicating program units, until the file is closed, either by the file input/output statement CLOSE, or by other statements, namely STOP and END, and the CHAIN statement when the FILES option is not specified.

File Input/Output Transmission Statements

GET #fileref Assigns values from a stream file or internal format file to a list of data items.

PUT #fileref	Writes values from a list of data items to a stream file.
READ #fileref	Retrieves a record from an internal file or native file and assigns values from it to a list of data items. Values from native files are formatted with a FORM statement or specification.
REREAD #fileref	Causes the record last read to be accessed again, and the data processed as in a READ #fileref statement. It is only valid for native files and values are formatted with a FORM statement or specification.
WRITE #fileref	Adds a record to a native, stream, or internal file. The data for the record comes from a list of data items; for native files, these values are formatted with a FORM statement or specification.
REWRITE #fileref	Alters a record already existing on a native file. A list of data items supplies the new values which are formatted with a FORM statement or specification.
DELETE #fileref	Removes a specific record from a keyed or relative file.
INPUT #fileref	Assigns values from files in three different ways. For display files, it functions as an INPUT statement; for internal files, it functions as a READ #file statement, and for stream files, it functions as a GET #file statement.
LINE INPUT #fileref	Assigns a complete record of unformatted data to a character variable.
PRINT #fileref	Transmits both formatted and unformatted data to a display file.

PROGRAM SEGMENTATION STATEMENTS

The program segmentation statements segment programs in several ways: through user-defined internal functions or subroutines, through external subprograms, or through external program chaining.

Internal functions or subroutines are defined by DEF/FNEND or GOSUB/RETURN statements, respectively. Each internal function or subroutine can be a logical entity within the program.

External subprograms are defined using the SUB, SUBEXIT, and END SUB statements. They are invoked with the CALL statement. These statements can be used to split a large program into manageable program units, consisting of a main program and one or more subprograms. The subprograms can be accessed (through the CALL statement) repetitively from the main program or from each other.

The CHAIN and USE statements can be used to split a very large program into several main programs (each, if needed, calling its own subprograms) so that each main program (through a CHAIN statement) can transfer control to the next at execution time.

In the CALL and CHAIN statements, values can be passed through arguments. Data can be shared by different programs in the COMMON area, which survives both a CALL and CHAIN statement execution.

The CALL statement can also access subprograms written in other languages. IBM BASIC supplies interface routines that allow the program to execute CMS commands, to perform Graphical Data

Display Manager/Presentation Graphics Feature (GDDM/PGF) operations, and to invoke subprograms written in COBOL, FORTRAN, or PL/I.

USER-DEFINED FUNCTION STATEMENTS

User-defined functions allow you to define new functions in addition to the intrinsic (built-in) functions already available to you. User-defined functions are specified through the following statements:

DEF Declares a user-defined function. It may define a numeric or character valued function.

The DEF statement may completely define the function or it may specify the beginning of a function block, or multiline function. The DEF statement is the first line of the block. It defines the function name and parameters

FNEND Marks both the physical and logical end of a multiline function. The FNEND statement is the last statement of the block. It serves the mark the end of block and is also the exit point of the function.

Functions are activated by a reference to the function name. You cannot transfer control into the body of a multistatement function.

A DEF statement, or a DEF/FNEND group of statements may appear anywhere in a program unit, except within another DEF/FNEND group.

Lines in a function definition are not executed unless the function they define is referenced. If execution reaches a DEF statement in some other fashion, processing proceeds to the line immediately following the function definition, bypassing the statements within the function.

When used in an executable statement, the function name may be followed by a list of arguments. This list must agree in number, order, and type with the list in the DEF statement.

A user-defined function can be:

- A single DEF statement containing an expression which determines the function's value.
- A multistatement function which is delimited by DEF and FNEND statements.

When a function is invoked, the arguments in the function reference, if any, are evaluated and their values assigned to the parameters in the parameter list for the function definition.

Transfer of control into or out of a function other than through function references is illegal. Unpredictable results may occur if input/output is performed by a function that has been invoked in an input/output data list, or if a function changes the value of a variable appearing in the same statement as the function reference.

A function name can be defined only once in a given program unit.

A function definition may not refer, directly or indirectly, to the function being defined, that is, recursive functions are not allowed.

A parameter appearing in the parameter list of a function definition is distinct from any variable with the same name outside the function definition.

Single Line Functions

If a function is completely defined in a DEF statement, the expression in that statement is evaluated and its value returned as the value of the function.

Multiline Functions

If a function is defined in a DEF block, the lines following the DEF line are executed in sequential order.

Within multiline functions, assignments (LET statements) of values to the function name determine the value to be returned as the value of the function.

Exit from a multiline function is accomplished by executing the FNEND statement.

The only exit from a multistatement function is through the FNEND statement. You cannot transfer control (for example, specify a GOTO statement) out of the body of a multistatement function.

Within a function, any GOSUB statement for which a matching RETURN statement has not been executed is removed from the RETURN list of GOSUB statements. Therefore, execution of a subsequent RETURN statement will not cause control to return into the function.

Processing a STOP statement in a DEF block ends the entire program (see "Subroutine Control Statements" on page 61).

SUBPROGRAM STATEMENTS

A program can be divided logically into a number of program units: a main program, and one or more subprograms.

Each program unit establishes a separate scope of identifiers. The same identifier may be used in different program units to name different items.

Statements within a program unit may not refer to any variable, array, line label, line number, or function (other than intrinsic functions) defined externally to that program unit.

Main Programs

A main program is a program unit whose first noncomment statement is any statement other than a SUB statement and whose last statement is an END statement.

CALL Passes control from the calling program to the specified subprogram. The arguments associated with the CALL must correspond to the parameters of the SUB statement for the program invoked.

A main program is the first program unit to receive control when processing is initiated. Other main programs may be invoked by means of the CHAIN statement (see "Chaining Statements" on page 81).

Subprograms

A subprogram begins with a SUB statement and ends with an END SUB statement. The SUB statement may be preceded by comment statements.

Subprograms are named in the SUB statement. They are invoked by calls (the CALL statement) from other program units (both main programs and other subprograms).

SUB Names the subprogram and names parameters which are variables to be used by the subprogram.

SUBEXIT Ends execution of a subprogram. This statement may occur only in a subprogram.

END SUB Marks the physical end of a subprogram and, if executed, ends execution of the subprogram.

A subprogram is a set of statements designed to perform a specific task. It might be a subprogram that can be used with more than one calling program to help solve several problems. For example, it might be necessary to write several programs, each of which must access and process a name and address file in the same manner. Processing the name and address file, then, is a prime candidate for becoming a subprogram.

It is possible for one program to access more than one subprogram, allowing data to pass not only to and from the main program, but between subprograms as well. A CALL statement is issued in a calling program in order to access a called subprogram. Called programs may in turn become calling programs. (See Figure 21.)

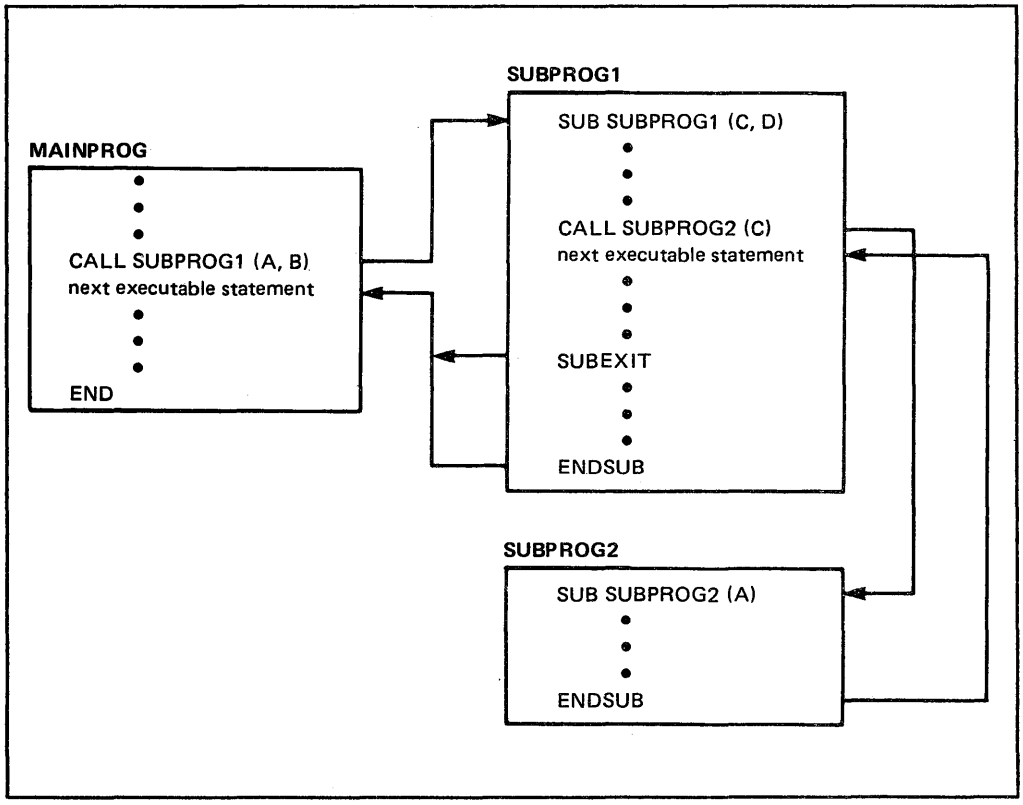


Figure 21. Calling and Called Programs

Subprogram A is both a called program and a calling program. It accepts data from subprogram B, processes it, and passes results back to the main program. Subprogram B is a called program, supplying data to both the main calling program and the calling subprogram A.

Arrays and character variables in a subprogram which are not parameters must appear in a dimension statement in that subprogram if they are to have other than the default dimensions or default maximum string lengths.

An array parameter is declared in the SUB statement with an "empty array declarator" which states how many dimensions the array has, but not the values of the dimensions. The actual values of the dimensions are those of the corresponding argument when the subprogram is called.

The same is true for parameters which are character variables. Both the current length and the maximum length of the parameter are passed as part of the CALL.

All arrays and variables which are not parameters and are not in COMMON are initialized to zero, for numerics, or null, for character strings, each time the subprogram is called.

Recursive subprogram calls are permitted.

Calling IBM BASIC Programs

The CALL statement passes control from the calling program to the subprogram specified.

The SUB statement is the first line of the subprogram, naming the subprogram and declaring any parameters.

When a CALL statement is executed, control transfers from the current program to the named subprogram. Execution of the subprogram begins at the line following the SUB statement in the called program and continues until:

- Some other action is dictated by execution of a control statement
- An error occurs that causes an abnormal termination
- An END SUB, STOP, or SUBEXIT statement is executed

The number and type of arguments in a CALL statement must agree with the number and type of parameters in the corresponding SUB statement. An array used as an argument must have the same number of dimensions as the corresponding parameter.

An array that is a parameter (that is, appears in a SUB statement) may be redimensioned within a subprogram. When control returns to the calling program, the array retains its changed dimensions.

See IBM BASIC Application Programming: System Services for more details.

Calling Programs Written in Other Languages

The CALL statement may be used to access subprograms written in other languages. However, because IBM BASIC's argument passing conventions differ from those of other languages, these calls must be made indirectly through interface routines which convert argument sequences.

BASIC follows general IBM calling conventions and generates object modules in standard IBM format. Modules created by other language processors may be linked with BASIC object modules. See IBM BASIC Application Programming: System Services for details.

Interface routines are supplied to establish linkage to routines written in COBOL, FORTRAN, and PL/I.

```
CALL COBOL (string expression, p1,p2,...)

CALL FORTRAN (string expression, p1,p2,...)

CALL PLI (string expression, p1,p2,...)
```

Where the value of string expression is the name of the routine to be called. p1,p2,... are arguments. See "CALL COBOL, FORTRAN or PLI Statement" on page 94 for argument conversion rules.

Note that because of the conversions, the called programs may not store back into decimal arguments.

Other considerations, particularly in the area of input/output, must be taken into consideration.

Calling the System

The supplied subprogram SYSTEM allows programs to execute host system commands. These commands are limited to those available for execution under program control (see Figure 43 on page 317). The syntax is:

```
100 CALL SYSTEM (string expression)
```

where the value of the string expression is a host system command.

This statement is the analog of the IBM BASIC SYSTEM command. If the host system detects errors, an exception occurs.

Calling the Graphical Data Display Manager (GDDM)

The Graphical Data Display Manager (GDDM) may be called to perform graphic operations. The syntax is

```
100 CALL GDDM (rcp, p1,p2,...)
```

Where rcp is a numeric expression whose rounded integer value specifies the GDDM request control parameter (RCP) for the operation to be performed. The allowable values and their corresponding operations are defined in the GDDM User's Guide.

p1,p2... are the actual arguments for the operation. The number and type of these arguments depends upon the request control parameter.

GDDM does not use IBM BASIC's floating decimal or varying length character string data types. The interface routine converts parameters of these types to single precision floating binary and fixed length character strings, respectively. Integer parameters are passed in their IBM BASIC format. For information on how to link to GDDM routines, see IBM BASIC Application Programming: System Services.

You cannot use the following GDDM functions:

```
FSEXIT
FSINIT
FSRNIT
FSTERM
SPINIT
```

These are GDDM initialization and completion functions handled automatically by BASIC's interface routine. See the GDDM User's Guide for a description of the functions that are available.

CHAINING STATEMENTS

CHAIN Ends execution of the current program (the chaining program) and starts another program (the chained

program). It also specifies which variables are to be passed from the chaining program to the chained program.

USE Specifies which variables the chained program is expecting to receive from the chaining program. A "by name" correspondence must exist: only those variables that have the same identifiers and attributes in both programs are passed.

The CHAIN and USE statements allow separate programs to be executed serially, without outside intervention. This capability is useful when segmenting very large programs.

Figure 22 shows how a chaining program might work:

1. The MENU program contains CHAIN statements to invoke other main programs which execute entirely independently of the MENU program.
2. When the chained programs complete execution, they chain back to the MENU program.

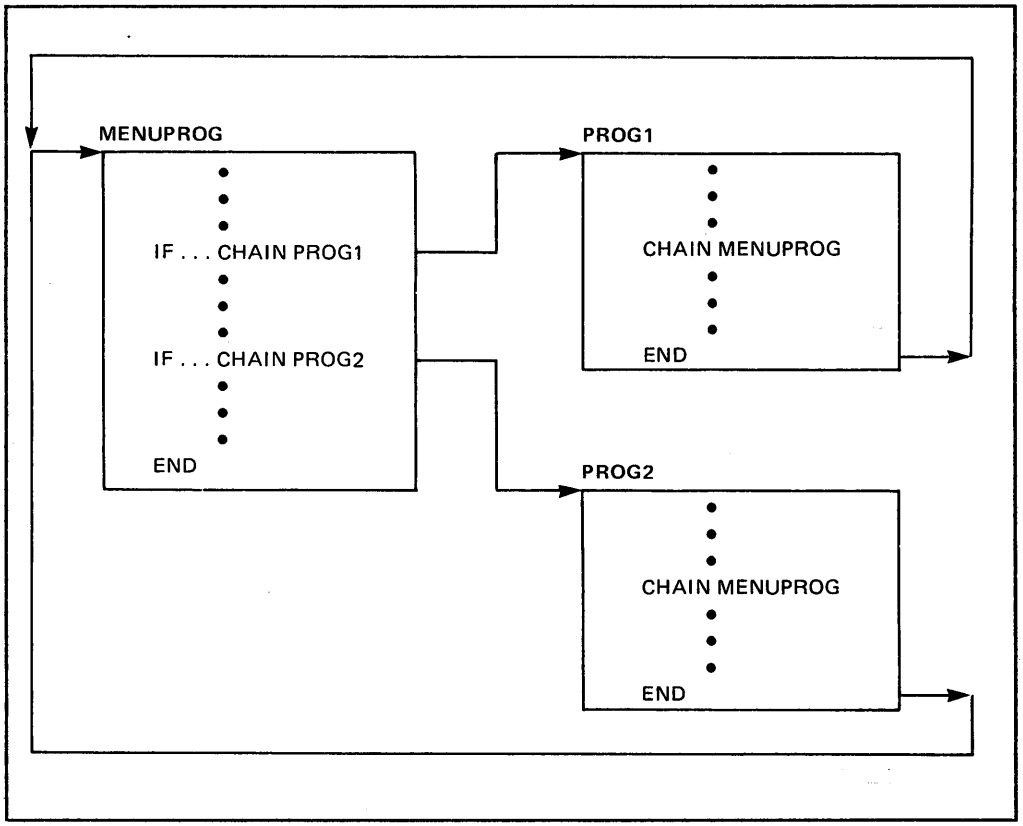


Figure 22. Chaining and Chained Programs

The CHAIN statement may optionally indicate whether the currently open files are to remain open or be closed prior to invoking the chained program.

The CHAIN and USE statements can be used to copy values from the chaining program to the chained program. The arguments are matched by name; if a name appears in only one list it is ignored. For names that do match, the type and size (for arrays and characters) are checked. If they match, the value is transferred; if not, an exception occurs.

PROGRAM SEGMENTATION RESTRICTIONS

CALL STATEMENT RESTRICTIONS: In the interactive mode, if the CALL statement is a source statement in the workspace, BASIC first checks to see if the subprogram is present in source form in the workspace. If it is, the workspace subprogram is used. Otherwise, BASIC attempts to find and load a compiled TEXT file with a filename the same as the subprogram name.

If the calling program is itself compiled, the subprogram must also be compiled. Thus a CALL within a dynamically loaded TEXT file cannot refer back to a program in the workspace.

CHAIN STATEMENT RESTRICTIONS: Main programs that are the targets of CHAIN statements are dynamically loaded. They can be compiled TEXT files or BASIC source program files.

When, in interactive mode, a CHAIN statement is encountered, BASIC first attempts to find a TEXT file of the indicated name. If a TEXT file is found, it is loaded and used. If no TEXT file is available, BASIC then attempts to find a source file, with the filetype of BASIC, and reload the workspace.

In programs running outside of the interactive BASIC environment, CHAIN statements can refer only to compiled TEXT files.

PROGRAM SEGMENTATION AND COMMON

Both the CALL and the CHAIN statements may explicitly pass arguments to another program unit using an argument list.

They may also pass arguments implicitly via the COMMON statement, which creates an area of storage that can be shared by many different programs.

A COMMON area created in a main program remains in existence from one subprogram CALL to the next as well as from one main program CHAIN to the next. Thus, a main program could declare a COMMON block, perform some actions which set values in COMMON, and then CHAIN to another main program which could continue to process those same values. See "COMMON Statement" on page 102 for more details.

EXCEPTION HANDLING STATEMENTS

Exception handling statements provide a means of regaining control in a program after an exception has occurred.

The exception handling statements are:

ON Condition Determines the action taken when an exception occurs: transfer control to a specified line number or label, ignore the exception, or perform the default action.

EXIT Specifies a line number or line label to which control is transferred when input or output exception conditions occur.

CAUSE Explicitly causes an exception; for example, for testing purposes.

CONTINUE Resumes execution after the statement which caused an exception.

RETRY Resumes execution after an exception by reexecuting from its beginning the statement which caused the exception.

USING I/O STATEMENT ERROR CLAUSES AND ON CONDITION STATEMENTS

The ON condition statement and error clauses within I/O statements (which may use EXIT statements) perform a similar function; the identification of what action the system should take if one of a general class of exceptions occur. The actions and their meanings are:

IGNORE Act as if the exception did not occur

GOTO Transfer program execution to a specified statement

SYSTEM Perform a default system action

These actions are explicit in an ON statement. In an I/O statement's error clause they are implicit; presence of a condition name implies GOTO; absence implies SYSTEM.

If an exception occurs, and control is transferred, whether via an error clause or ON statement, the following intrinsic functions are available for processing the exception:

CODE Obtains the system error code

ERR Obtains the IBM BASIC exception code

LINE Obtains the statement line number of the exception

EXCEPTION HANDLING IN I/O STATEMENTS

Error clauses in I/O statements take priority over the ON condition statement. That is, when an exception occurs, the processor first checks if an applicable error clause is specified in the I/O statement. If it is, control is transferred as specified by the error clause. If the exception does not correspond to any error clause, the action taken is determined from the ON condition statement, if any.

The error conditions, ENDPAGE, CONV, and SOFLOW, are equivalent to the ON conditions of the same name. All other error clause or EXIT conditions are equivalent to the ERROR condition of the ON statement.

USING THE CAUSE STATEMENT

The user can generate an exception explicitly with the CAUSE statement. This can be used to test routines that handle abnormal conditions.

USING THE RETRY AND CONTINUE STATEMENTS

Use the RETRY and CONTINUE statements to resume program execution:

- RETRY—resumes execution with the statement that caused the exception
- CONTINUE—resumes execution with the statement following that which caused the exception

In general, any statements in the language may be used to attempt recovery from the error condition. If another exception occurs, and IBM BASIC has not been told to IGNORE it, all knowledge of the first exception is lost. In this case, a RETRY or CONTINUE statement resumes execution at or after the location of the second exception.

EXCEPTIONS AND USER-DEFINED FUNCTIONS

The relationship between exceptions and user-defined functions requires some additional explanation. An EXIT or ON condition statement may indicate that the processor should GOTO any line number or label in the program unit without restriction. When an exception occurs, the following rules are followed:

- If the indicated statement is not in the main body of the program unit or in a currently executing user function, the exception message is written and an "invalid exception location" exception is generated. Otherwise:
  - Each executing function is immediately exited, that is, the remaining code is not executed until the function (or main body of the program unit) containing the GOTO destination is reached.
  - The intrinsic function is set to the statement containing the function invocation at the now current level.
  - Control is transferred to the designated line.

  Example

```
100 ON OFLOW GOTO 360
110 DEF A(P1)
120     A = X/P1
130 FNEND
210 DEF B(P2)
220     ON ZDIV GOTO 250
230     B = P2*A(P2)
240     GOTO 260
250     PRINT 'ZDIV AT LINE';LINE
260 FNEND
310 X = 1.E50
320 Y = B(0)
330 Y = B(1.E-50)
340 Y = A(0)
350 STOP
360 PRINT 'OFLOW AT LINE';LINE
370 CONTINUE
380 END
```

In this example, the first invocation of B (at line 320) causes an exception at line 120 in A. A is exited, and execution will resume at line 250 and LINE will be set to 230.

The next invocation of B (at line 330) will cause an exception at line 120 in A as well. Both A and B will be exited; execution will resume at line 360, and LINE will be set to 330. The CONTINUE statement will transfer control to line 340. This too will cause an exception at line 120 in A. However, the ON ZDIV still points to line 250 in B, which is not active, hence a ZDIV error message will be written and an "invalid exception location" exception generated.

Note: This exception can be handled with an ON ERROR clause.

EXCEPTIONS AND CALLING AND CALLED PROGRAMS

Each program unit (subprogram or main program) has a separate set of EXIT and ON condition lists. If one program unit calls another, it loses all control over any exceptions until the called program unit returns. At that time the EXIT and ON conditions are reset to their state prior to the call.

DEBUGGING STATEMENTS

Debugging facilities allow the user to build test points into programs. With debugging statements, the user can set breakpoints, trace the execution of a program, and turn the debugging system ON and OFF.

DEBUG ON/OFF The DEBUG ON statement causes debugging to become active in the program unit in which it is specified.

The DEBUG OFF statement causes debugging to become inactive in the program unit in which it is specified. If a DEBUG OFF statement is executed when tracing is in progress, an implicit TRACE OFF statement is executed; that is, when a subsequent DEBUG ON statement is executed tracing does not resume.

Before the execution of any debug statement in a program unit, debugging is inactive (OFF).

BREAK The BREAK statement, when debugging is active, reports the line number of the BREAK statement and suspends processing. (This is called a breakpoint.) At this time, the user can continue execution by pressing the ENTER key, or can enter IBM BASIC commands and immediate statements before continuing.

If the program is not modified, processing can be resumed by issuing the GO command.

If the program is modified in any way, processing cannot be resumed at the breakpoint; instead, the program must be reinitiated via the RUN command. Thus, any line number editing or use of the following commands end execution: CHANGE, COMPILE, COPY, DELETE, DROP (of any program variables), EXTRACT, FETCH, INIT, LOAD, MERGE, RENUMBER, RUN.

A BREAK statement is ignored when debugging is inactive.

TRACE ON/OFF The TRACE ON statement, when debugging is active, turns tracing on in the program unit in which it is specified.

The TRACE OFF statement, when debugging is active, turns tracing off in the program unit in which it is specified.

Before the execution of any TRACE statement in a program unit, tracing is set off.

A TRACE statement is ignored when debugging is inactive.

USING THE TRACE STATEMENT

The following actions occur when tracing is on.

- For each statement causing a transfer of control (for example, a GOTO, or CALL, or NEXT), both the line number of the statement and the line number of the next statement to be processed (if such a line number exists) are reported.
- For each statement causing the value of a variable or array to change, both the line number of the statement and the values assigned to any variables by the statement are reported.

Trace reports may be directed to files by means of the TO clause in the DEBUG ON and TRACE ON statements.

Four rules govern the use of the TO clause.

1. A TO clause in a DEBUG ON statement overrides any TO clauses of TRACE ON statements encountered subsequent to the DEBUG ON statement in the same program unit.
2. A TRACE OFF statement breaks the file connection set by any TRACE ON TO statement in the same program unit.
3. A TRACE ON TO statement breaks the file connection established by a prior TRACE ON TO statement in the same program unit. It does not break the file connection established by a prior DEBUG ON TO statement.
4. A DEBUG OFF statement breaks the file connection as well as trace output established by any prior DEBUG ON TO or TRACE ON TO statement in the same program unit.

If no file reference is specified, the trace report is directed to the device associated with file reference zero (usually the terminal).

IMMEDIATE STATEMENTS AND DEBUGGING

Statements that can be executed in the immediate mode (although they are not classified as debugging statements) are very useful for program debugging. For example, while stopped at a breakpoint you can execute an immediate PRINT statement to examine the value of any variable in the active program unit.

STATEMENT DESCRIPTIONS

This section describes the syntax and semantics of each statement in the IBM BASIC language. The statements appear in alphabetic order. For the relationships between statements, see the previous sections in the chapter on "IBM BASIC Statements" on page 60.

BREAK STATEMENT

The BREAK statement (when debugging is active) suspends program execution, identifies the current line, and makes possible interaction with the system. The suspension of execution is known as a breakpoint.

Execution of a BREAK statement when debugging is inactive has no effect.

Format

```
BREAK
```

Description

The BREAK statement provides a means of using the debugging capabilities of IBM BASIC.

System commands may be used without inhibiting processing, if they do not modify the program. If the program is modified in any way, execution cannot be resumed at the breakpoint and must be reinitiated via the RUN command. Thus, any line number editing or use of the following commands ends processing: CHANGE, COMPILE, COPY, DELETE, DROP (of any program variables), EXTRACT, FETCH, INIT, LOAD, RENUMBER, RUN.

If the program is not modified, enter either a GO or a null entry to restart program execution after a BREAK statement. (A null entry can be used only if no other commands have been entered while at the breakpoint.)

The BREAK statement is controlled by the DEBUG statement. See "Debugging Statements" on page 86 and "DEBUG Statement" on page 106.

See "BREAK Command" on page 271 for another method of causing a program break (without the necessity of editing and rerunning your program).

CALL STATEMENT

The CALL statement invokes subprograms.

Format

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

Where:

name
identifies the subprogram to be run. Subprogram names may contain at most seven characters.

If the name is SYSTEM, COBOL, FORTRAN, PLI, or GDDM, see the special format on the following pages.

arg
is an argument passed from the calling program to the called subprogram. It is an expression (numeric or character) or an array from the calling program that may be accessed by the called subprogram.

When an entire array is to be passed as an argument, it must be stated as an empty array declarator to indicate the number of dimensions in the form:

```
identifier ([,]...)
```

Description

When a CALL statement is executed, control passes from the calling program to the named subprogram. When the subprogram completes execution, control returns to the calling program at the next executable statement after the CALL statement.

Example

```
100 REM MAIN PROGRAM STATEMENT
110 CALL DEDUCT
120 REM NEXT STATEMENT OF MAIN
.
.
.
600 SUB DEDUCT
.
.
.
700 END SUB
```

The CALL statement at line 110 passes control to the subprogram DEDUCT, identified by the SUB statement. Processing continues until the END SUB statement signals a return to the main program. Execution continues at the first statement following the CALL (statement 120).

The number and type of arguments used in the CALL statement must agree with the number and type of parameters in the corresponding SUB statement. An array used as an argument must have the same number of dimensions as the corresponding parameter in a SUB statement.

Arguments that are numeric variables or character variables (without substring qualifiers) are passed by reference, as follows:

- Any reference to such a parameter in a subprogram is a reference to the corresponding argument in the calling program.
- Any assignment to such a parameter in a subprogram is an assignment to the corresponding argument in the calling program.

If an argument is an array element, its subscripts are evaluated once, when the CALL statement is executed. The previously stated rules apply.

If an argument is a constant or an expression that involves numeric or character operators, it is evaluated once, when the CALL statement is executed. (Note that a character substring is considered such an expression.) The evaluated value is assigned to a temporary location available only to the subprogram. In any reference to the corresponding subprogram parameter, this temporary value is used; in any assignment to the corresponding subprogram parameter, this temporary location is used.

See also "Calling IBM BASIC Programs" on page 80.

Predefined Subprogram Names

The CALL statement can be used to call programs written in other languages (COBOL, FORTRAN, or PL/I), to request operations from the Graphical Data Dislay Manager (GDDM), or to execute host operating system commands (SYSTEM). This is done by specifying special subprogram names in CALL statements. (See "CALL SYSTEM Statement" on page 92, "CALL GDDM Statement" on page 93, and "CALL COBOL, FORTRAN or PLI Statement" on page 94.) The special subprogram names are:

CLINK	GDDM
COBOL	PLINK
FLINK	PLI
FORTRAN	SYSTEM

They are keywords and, as distributed by IBM, they are also reserved words.

If your organization has removed these names from the reserved word list, you can use them as variable or array names or as line labels. However, because of their keyword meanings, you cannot use them in SUB statements to name subprograms. If used in the CALL statement, they will always refer to the predefined subprograms.

CALL SYSTEM Statement

The CALL SYSTEM statement allows you to execute a limited set of system commands.

Format

```
CALL SYSTEM (string expression)
```

Where:

string expression
must evaluate to a character string that is a system command.

Description

The commands available for use with CALL SYSTEM are limited to those available for execution under program control. See Figure 43 on page 317 for a list of the commands available. Be careful when using these commands; some of them can adversely affect your BASIC terminal session.

When the CALL SYSTEM statement is executed, the system command in string expression is executed. If the command displays information at the terminal, and the terminal is a display terminal, the BASIC screen is temporarily replaced by the system screen. See IBM BASIC Application Programming: System Services for methods of restoring the BASIC screen.

If an error occurs during command execution, an exception occurs.

See also "Calling the System" on page 81.

CALL GDDM Statement

The CALL GDDM statement is used to perform graphic operations.

Format

```
CALL GDDM (rcp,a1,a2,...)
```

Where:

rcp
is a numeric expression whose rounded integer value specifies the Request Control Parameter for the Graphic Data Display Manager (GDDM).

a1,a2,...
are arguments for the operation.

Description

The allowable values and their corresponding operations are defined in the GDDM User's Guide.

Additional details are given in IBM BASIC Application Programming: System Services.

See also "Calling the Graphical Data Display Manager (GDDM)" on page 81 for a discussion of parameter passing rules.

CALL Statement

CALL COBOL, FORTRAN or PLI Statement

The CALL COBOL, FORTRAN, or PLI statements are interface routines that establish linkage to routines written in COBOL, FORTRAN, or PL/I.

Format

```
CALL {COBOL|FORTRAN|PLI}
     (string expression,a1,a2,...)
```

Where:

string expression
evaluates to a character string that is the name of the routine to be called

a1,a2,...
are arguments that are converted as shown in Figure 23.

Description

Because of differences in internal data representations, the called programs must return values to IBM BASIC carefully. Only integer and character parameters may be passed back to IBM BASIC and they must conform to the characteristics of IBM BASIC, that is, fullword integers and appropriate string lengths.

Entire arrays cannot be used as arguments in interlanguage calls.

Decimal values cannot be returned by the called program. Integer and character values can be returned, but care must be taken with character string lengths. When calling COBOL, the current length (at the time of the call) of a character argument must be equal to m in Figure 23. When calling PL/I, the current length must be less than or equal to n in Figure 23 and, if a value is to be returned, the maximum length defined for the BASIC variable must be equal to n.

Before it can call COBOL, FORTRAN, or PL/I, IBM BASIC must be told which programs are to be called in a CALL CLINK, CALL FLINK, or CALL PLINK statement, respectively. IBM BASIC Application Programming: System Services gives details.

See also "Calling Programs Written in Other Languages" on page 80.

BASIC	COBOL	FORTRAN	PL/I
INTEGER	PIC S9(9) USAGE COMP-4	INTEGER	FIXED BIN (31)
DECIMAL	USAGE COMP-2	REAL*8	FLOAT DEC (16)
CHARACTER	PIC X(m) USAGE DISPLAY	not allowed	CHAR(n) VARYING

Figure 23. Type Conversions for Interlanguage Calls

CASE STATEMENT

The CASE statement immediately precedes a group of statements (a CASE block) within a SELECT block that are executed when the value of the selection expression in the SELECT statement satisfies the criteria of the CASE statement. The group of statements is referred to as a CASE block.

Format

```
CASE selector [,selector]...
```

Where:

selector
is one of the following:

```
constant

constant TO constant

relation constant
```

and:

constant
is a constant of the same type, either numeric or character, as the selection expression for the containing SELECT block.

relation
is one of the relational operators.

Description

The CASE statement is used with the SELECT, CASE ELSE and END SELECT statements to form a CASE block within a SELECT block.

CASE blocks include all statement lines between a CASE statement and either the next CASE statement, CASE ELSE statement, or an END SELECT statement.

The CASE statement may appear only within a SELECT block. It defines the beginning of a CASE block and the selection criteria for that block.

The constants and relations on a CASE statement define which CASE block will be executed when the selection expression of the SELECT statement is evaluated.

See also "SELECT Blocks" on page 66.

Example

```
100 CASE <0, 50 TO 60, >100
```

specifies this CASE block will be selected for any numeric values less than zero, for any value between 50 and 60, inclusive, and for any value greater than 100.

```
200 CASE 'MAY','JUNE','JULY'
```

specifies this CASE block will be selected for a character value of 'MAY', 'JUNE', or 'JULY'

CASE ELSE Statement

CASE ELSE STATEMENT

The CASE ELSE statement immediately precedes a group of statements (a CASE block) within a SELECT block that are executed if no other CASE blocks are selected.

Format

```
CASE ELSE
```

Description

The CASE ELSE statement defines the CASE block to be executed if selection criteria for the other CASE blocks are not met.

The CASE ELSE statement is used with the SELECT, CASE, and END SELECT statements to form SELECT blocks. These are discussed under "Decision Structure Control Statements" on page 64.

The CASE ELSE statement may only appear within a SELECT block. It must begin the last CASE block in a SELECT block.

See also "SELECT Blocks" on page 66.

CASE ELSE Statement

CASE ELSE STATEMENT

The CASE ELSE statement immediately precedes a group of statements (a CASE block) within a SELECT block that are executed if no other CASE blocks are selected.

Format

```
CASE ELSE
```

Description

The CASE ELSE statement defines the CASE block to be executed if selection criteria for the other CASE blocks are not met.

The CASE ELSE statement is used with the SELECT, CASE, and END SELECT statements to form SELECT blocks. These are discussed under "Decision Structure Control Statements" on page 64.

The CASE ELSE statement may only appear within a SELECT block. It must begin the last CASE block in a SELECT block.

See also "SELECT Blocks" on page 66.

CAUSE STATEMENT

The CAUSE statement generates an exception during processing.

Format

```
CAUSE numeric expression
```

Where:

numeric expression
is any numeric expression.

Description

Exceptions are normally generated implicitly when error conditions arise. The CAUSE statement may be used to explicitly create an exception.

The expression is evaluated and, if decimal, rounded and converted to integer. The result is used as the exception code.

Exception codes are listed in "Appendix A. Exception Codes" on page 319. You are not limited to this set of codes. Any codes not listed there are treated as ERROR category exceptions (see "ON Condition Statement" on page 203).

See also "Exception Handling Statements" on page 84.

Example

```
      .
      .
      .
  200 ON ERROR GO TO 500
      .
      .
      .
  500 IF ERR = -7320 THEN FILE_NOT_FOUND
  510 ON ERROR SYSTEM
  520 CAUSE ERR
```

If, during program execution, an ERROR exception occurs, control is transferred to line 500. At line 500 the exception for file-not-found (-7320) is tested, and, if this is the ERROR exception, control is transferred to the line with label FILE_NOT_FOUND.

If some other exception has occurred, the CAUSE statement is executed, forcing the system action for that exception to take place. (The ERR intrinsic function returns the exception code.)

CHAIN STATEMENT

The CHAIN statement halts processing of the program in which it appears and starts a new main program.

Format

```
CHAIN char-exp [,FILES][,arg-list]
```

Where:

char-exp
is a character expression naming the chained program (the name of the file containing the new main program being started).

arg-list
is a list of variable and/or array names separated by commas.

Description

When a CHAIN statement is executed, the program in process is terminated and the program named in the CHAIN statement is invoked.

FILES KEYWORD: The optional keyword FILES may follow the chained program name. If it is present, all currently open files remain open, and at their current position. If the keyword FILES is not present, all files in the chaining program are closed when the CHAIN statement is executed.

ARGUMENT LIST: A list of data items may follow. The list defines the names of the variables and arrays in the chaining program that are to retain their current values when the chained program begins executing. If the CHAIN statement is specified in a subprogram, the argument list may not include the names of COMMON variables or parameters. All other data except COMMON variables are destroyed during the chaining operation.

This list is compared against the identifiers listed in USE statements within the chained program. Only those identifiers that match (have the same name, type, and dimensions) are passed to the chained program.

GENERAL CONSIDERATIONS: Chaining allows separate programs to be processed serially, without outside intervention. This procedure is useful when segmenting large programs (that is, breaking them into smaller, more manageable pieces).

A CHAIN statement may not be used in a function definition (between DEF and FNEND statements).

See also "Program Segmentation Statements" on page 76 "CHAIN Statement," and "USE Statement" on page 257.

Example

Program X contains this statement

```
100 CHAIN "MYFILE",FILES,C,B,A
```

Program MYFILE contains this statement

```
200 USE A,B,C
```

This example keeps all of program X's currently open files open, but stops processing program X and starts processing MYFILE. The values of variables A, B, and C are passed to MYFILE from X. The variables A, B, and C must be the same type in both program X and program MYFILE. If variable A is typed integer in program X and decimal in program MYFILE, an exception occurs.

CLOSE Statement

CLOSE STATEMENT

The CLOSE statement deactivates the specified file.

Format

```
CLOSE #fileref : [err[,err]]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 0 to 255.

err
is one of the following:

EXIT line-ref

IOERR line-ref

EOF line-ref

line-ref
is a line number or line label

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

The colon after fileref may be omitted if it is the last nonblank character on the line.

Example

```
300 CLOSE #2
```

Description

The CLOSE statement prevents further successful access to the file until another OPEN statement is processed for that file. An attempt to close a file that is not open results in an exception.

A CLOSE statement issued for a display format file which has an incomplete print line waiting to be written (that is, the last PRINT statement ended with a comma or semicolon) causes the print line to be written before the file is closed.

The STOP, END, and CHAIN statements (without the keyword FILES) automatically close all active files. Any resulting errors are handled as if they were caused by a CLOSE statement.

FILEREF: Fileref is the reference number of the file to be closed.

An attempt to close fileref 0, the system device, is ignored and a warning message is produced.

ERROR CONDITIONS: The two error conditions EOF (end of file) and IOERR (input/output error) may be recoverable if an err clause for the condition is specified in the statement or on the referenced EXIT statement. EOF occurs if the file cannot be closed because of lack of space. IOERR occurs if a hardware malfunction or other condition prevents closing of the file.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 CLOSE #5: EXIT 200
200 EXIT IOERR 500, EOF 800
```

is functionally equivalent to:

```
100 CLOSE #5: IOERR 500, EOF 800
```

In this example, if the file associated with file reference number 5 cannot be closed because of a hardware malfunction, control is transferred to line number 500. If it cannot be closed because of a lack of space on the file, control passes to line number 800.

COMMON STATEMENT

The COMMON statement provides a means of sharing values between either a main program and subprograms, or between main programs when a CHAIN statement is executed.

Format

```
COM[MON] m1[,m2]...
```

Where:

m1, m2 ...
may each be one of the following:

a
b[*L]

and where:

a
is a numeric variable or array declarator.

b
is a character variable or array declarator.

L
is an unsigned integer constant between 1 and 32,767, giving the maximum length of the character variable.

array declarator
has the form c(d1[,d2[,d3[,d4[,d5[,d6[,d7]]]]]]) where c is a numeric or character identifier and each d is an unsigned integer constant in the range 0 or 1 (depending upon OPTION BASE) to 32767.

Description

The COMMON statement explicitly defines the number and extents of array dimensions and the maximum string length of character variables and array elements, and allocates these variables and arrays in a common area. The array dimensions and string lengths are specified as they are in the DIM statement.

In order to access the common area, a program unit must include a COMMON statement. COMMON statements may appear anywhere within a program unit. Items in common are allocated in the order of their appearance. Variables and arrays that are defined in COMMON statements may not also appear in DIM statements.

The common area is initialized once, upon entry to the first program unit defining the common area; all numeric items are set to zero and character variables are set to null. When the first program unit using common is executed, the size of the common area is determined; subsequent program units may specify a common area of an equal or smaller size, but they may not extend the common area.

Program units that share the common area must agree in their image of the common area. This means that, although they need not have the same names, the common variables and arrays must be declared in the same order and with the same characteristics; variables must be of the same type, character variables and array elements must have the same maximum string length, and arrays must be of the same total size.

Arrays can be declared with different dimensions as long as the total size remains the same. The sizes are checked when a common array is passed between program units, but the array's dimensions are left as they exist in the calling program. The converse is also true; if a called subprogram redimensions a common array, the new dimensions are returned to the caller.

The following examples illustrate correct and incorrect usage of COMMON.

Example 1 (correct)

```
100 COM A,B%(3,3),C$(16)*10
110 COM D#(5)
```

creates a common area with the following four items: a decimal variable named A, a 4-by-4 integer array named B%, a 17-element, one-dimensional character array named C$ with each element having a maximum length of 10 characters, a 6-element, one-dimensional decimal array named D#.

Example 2 (incorrect)

```
100 COM A
110 A(3) = PI
```

causes an exception because common arrays must be explicitly dimensioned.

Example 3 (incorrect)

```
100 COM A
110 DIM A(100)
```

causes an exception because arrays in common must be dimensioned in the COMMON statement, not by a DIM statement.

Example 4 (incorrect)

```
100 COM A, C$(99)*3, B%(1,3),D
110 CALL JOE
120 END
130 SUB JOE
140 COM W,Y$(99)*6,X%(0,7)
150 X%(0,6)=59
160 END SUB
```

contains two errors. First, C$ and Y$ have different maximum string lengths; second, line 150 does not agree with the dimensions of the array at the time of the call. The following changes result in a correct program:

```
105 MAT B%=B%(0,7)!redimension B%
140 COM W,Y$(99)*3,X%(0,7)
```

or, you could code:

```
100 COM A,C$(99)*6,B%(1,3),D
145 MAT X%=X%(0,7)!redimension X%
```

CONTINUE Statement

CONTINUE STATEMENT

The CONTINUE statement is used to resume execution at the statement following the statement causing an exception.

Format

```
CONTINUE
```

Description

The CONTINUE statement provides for the return to normal sequential statement execution after program flow has been diverted to process an exception.

Assume that you wanted to keep track of the number of times in your program that an attempt was made to divide a number by zero. You did not want to halt the program on this error, just count the occurrences.

Example

```
100 ON ZDIV GOTO 1000
.
.
.
500 BAL = A - B
510 DIVI = TOT/BAL
520 BAL = A + B
.
.
.
1000 COUNT = COUNT + 1
1010 CONTINUE
```

Statement 100 sets the condition being tested. If BAL is set to zero at statement 500, execution of 510 triggers the ZDIV (divide by zero) condition. Execution branches to statement 1000, adds 1 to COUNT, and returns to statement 520 because of the CONTINUE.

If an exception condition does not exist when CONTINUE is executed, an exception occurs.

See also "Exception Handling Statements" on page 84.

DATA STATEMENT

The DATA statement is used to create internal data files for reference by READ statements.

Format

```
DATA [integer*]item [,[integer*]item]...
```

Where:

integer
is a nonzero, unsigned integer constant used to replicate the immediately following item.

item
is either a constant (either numeric or character) or an unquoted character string.

Description

DATA statements are nonexecutable statements which are used to create a data file internal to the program unit. They can appear anywhere in the program unit, but all of the DATA statements create one internal file of values whose order is determined by the line numbers of the statements.

Both character and numeric constants can be used in DATA statements. A character constant may be specified without surrounding quotation marks provided the constant does not start with an integer immediately followed by an asterisk and provided the constant contains no commas, no leading or trailing blanks, and no leading or trailing quotes. A valid numeric constant may be assigned to either a numeric or a character variable; however, if a numeric constant is assigned to a character variable, it is assigned as a string of characters.

Specifying the replication factor (for example, 10*MOB$) is equivalent to repeating the variable MOB$ 10 consecutive times.

See also "READ Statement" on page 235.

Example

```
100 DATA 3*10.0,"APPLES",2.5,PEARS,19
200 DATA PEACHES,2*24,2*BANANAS
```

The above DATA statements create an internal data file which, when accessed by READ statements, provides the following sequence of values:

```
10.0
10.0
10.0
APPLES
2.5
PEARS
19
PEACHES
24
24
BANANAS
BANANAS
```

DEBUG Statement

DEBUG STATEMENT

The DEBUG statement activates debugging facilities in a program.

Format

```
DEBUG ON [TO #fileref]
    or
DEBUG OFF
```

Where:

fileref
is a numeric expression, the rounded, integer value of which must be in the range of 0 to 255, specifying the file for trace listing.

Description

Debugging facilities are provided by language statements in order to allow test points to be built into a program.

The DEBUG statement allows you to turn the debugging facility ON and OFF within each program-unit. The DEBUG statement acts as an ON/OFF access switch:

- DEBUG ON causes debugging to become active, making the statements BREAK and TRACE available for use.
- DEBUG OFF causes debugging to become inactive. In a program unit, before the processing of a DEBUG statement, debugging is inactive. TRACE and BREAK statements have no effect when debugging is inactive.

The fileref in the optional TO clause overrides subsequent TO clauses in TRACE statements.

See "Debugging Statements" on page 86.

Immediate Execution

The DEBUG statement may be executed as an immediate statement. All forms are accepted in the immediate mode. However, if the program unit did not contain both a TRACE ON and a DEBUG ON statement prior to the start of execution, the trace facility, when activated by TRACE ON, monitors program flow only; it does not show variable assignments.

See also "Immediate Statements" on page 260.

DECIMAL STATEMENT

The DECIMAL statement specifies which identifiers are to be assigned decimal type.

Format

```
DECIMAL [[identifier|(letter-list)]...]
```

Where:

identifier
may be a specific numeric identifier.

letter-list
is a list of letters and/or ranges of letters separated by commas. A range of letters is represented by the first and last letters in the range separated by a minus sign.

For compatibility with other BASICs, the reserved words DEFSNG (define single) and DEFDBL (define double) may be used in place of the keyword DECIMAL. The syntax and semantics of DEFSNG and DEFDBL statements are the same as for DECIMAL statements.

Description

The DECIMAL statement declares a specific identifier or any identifier beginning with a specific letter as having decimal type, or when used without a list, to specify the default type for all identifiers not otherwise typed in a program unit. DECIMAL statements may appear anywhere in a program unit, and affect identifiers throughout the program unit. The identifiers affected may be variable names, array names, or function names.

An identifier explicitly stated in a DECIMAL statement may end with the self-typing character "#", but not with "%" or "$". If the DECIMAL statement specifies a parenthetical list of letters, all identifiers beginning with these letters are to be typed decimal, unless they end in a contradictory self-typing character "%" or "$", or unless they are explicitly declared in an INTEGER statement by identifier or letter-list. The letter-list may be specified as either single letters (A, B, D, J) or as a series of consecutive letters, such as (A-J, T-Z), indicating A through J and T through Z.

If a DECIMAL statement specifies no identifiers and no letter-list, the default type for all identifiers in the program unit is set to decimal. This is the default.

Example 1

```
100 DECIMAL ABLE,(C-E,G,J,L),NANCY
```

specifies that identifiers ABLE and NANCY, as well as all identifiers beginning with the letters C, D, E, G, J, and L are typed decimal. If the program unit subsequently contains a variable named DANDY, it would be assigned decimal type; however, COLOR$ would be character and LOT% would be integer.

Example 2

```
100 DEFSNG ABLE,(C-E,G,J,L),NANCY
```

is equivalent to Example 1.

DECIMAL Statement

Immediate Execution

You can use the DECIMAL statement to set the type of immediate variables and arrays. The format and description are the same as for a DECIMAL statement in a program.

See "Immediate Statements" on page 260 and "Immediate Type and Dimensions" on page 262 for the rules regarding the interaction with other immediate statements and program statements.

DEF STATEMENT

The DEF statement defines and names a user-written function.

Format

```
DEF name [(param[,param]...)] [=expression]
```

Where:

name
is a scalar numeric or character name that gives a name to the function and its input. If it is character (ends with $), the maximum length may be defined by following the identifier with an asterisk and an integer (between 1 and the maximum allowed string length).

param
is a scalar numeric or character variable name that specifies the function input. If it is character (ends with $), the maximum length may be defined by following the identifier with an asterisk and an integer.

expression
is an expression of the same type, numeric or character, as name, used to complete a one-line function definition.

Description

The DEF statement is a nonexecutable statement that defines user functions. The user function definition may be contained in the DEF statement itself by including the equal sign and expression. Otherwise, the DEF statement marks the beginning of a group of statements, ending with an FNEND statement, which constitutes the function definition.

A user function is referred to in other statements within the same program unit in a manner similar to the intrinsic functions. When used in an executable statement, the function name is optionally followed by a list of arguments, separated by commas and enclosed in parentheses. This list of arguments must agree in number, order, and type with the list of parameters in the DEF statement.

Example

```
100 DEF E_TO_X_SQUARED(X) = EXP(X**2)
```

defines the natural exponential of X squared, using the intrinsic function EXP. The numeric variable X, enclosed in parentheses after the function name, is called a parameter. You can have more than one parameter, and the list of variables can contain both numeric and character variables. Your function performs its defined calculation on the actual values supplied for these parameters. (The expression value substituted for each parameter is called the argument.)

Example

```
220 ANS=E_To_X_SQUARED(5)
```

The value 5 is substituted for the parameter X.

A DEF statement or DEF/FNEND group of statements may appear anywhere in a program unit, except within another DEF/FNEND group.

A user defined function may be executed only through a function reference. Any other transfer to the DEF statement results in a transfer to the statement following the function.

Transfer of control into or out of user defined functions, other than through function references, is illegal.

Nonexecutable statements, such as COM, DATA, DECIMAL, DIM, INTEGER, USE, EXIT, FORM and IMAGE, may appear within a user-defined function. A user-defined function can refer to or change values of any variable, except those used as parameters in the function, in the program unit containing the function.

Undefined results may occur if:

1. A user defined function performs any input/output, and the function reference has been involved in an input/output data list.

2. A user defined function changes the value of a variable appearing in the same statement as the function reference.

A function of a given name can be defined only once in a given program unit.

When a defined function is referenced (that is, when an expression involving the function is evaluated), the arguments in the function reference, if any, are evaluated and their values are assigned to the parameters in the argument list for the function definition (that is, arguments are passed by value to functions).

A function definition may not refer, directly or indirectly, to the function being defined; that is, recursive function invocations are not permitted.

A parameter appearing in the parameter list of a function definition is local to that function definition; that is, it is distinct from any variable with the same name outside the function definition.

SINGLE LINE FUNCTIONS: If a function is completely defined in a DEF statement, the expression in that statement is evaluated next and its value assigned as the value of the function.

MULTILINE FUNCTIONS: A function defined over many statements is called a multiline function. A multiline function begins with the word DEF, the function name, and any parameters, the same as single-line functions.

Within a multiline function definition, an assignment to the function name establishes the value returned when that function evaluation is complete.

If the flow of control through a multiline function is such that the function name is not assigned a value, the value returned is the value returned by the previous invocation of the function. If the function has not been previously invoked, zero or a null string is returned for numeric or character functions, respectively.

The FNEND statement indicates both the physical and logical end of a multiline function.

If a function is defined in a DEF block (a multiline function), the lines following the DEF line are processed in sequential order until:

- Some other action is dictated by processing of a control statement
- An exception occurs
- An FNEND or STOP statement is executed

Execution of a STOP statement in a DEF block ends processing of the entire program. See "FNEND Statement" on page 126.

Example

```
100 DEF POSITIVE_DIFFERENCE(X,Y)
110 IF X>Y THEN
120    POSITIVE_DIFFERENCE = X-Y
130 ELSE
140    POSITIVE_DIFFERENCE = Y-X
150 END IF
160 FNEND
```

DELETE File Statement

DELETE FILE STATEMENT

The DELETE File statement causes deletion of a record from a keyed or relative file.

Format

```
DELETE #fileref [,] pos: [err[,err]...]
```

Where:

fileref
is a numeric expression that, when evaluated and rounded, must be a positive integer within the range 1 to 255.

pos
is KEY [=|EQ] character expression

or

REC[ORD] [=|EQ] numeric expression

Note: fileref and pos may appear in any sequence.

err
is one of the following:

EXIT line-ref

NOREC line-ref

NOKEY line-ref

IOERR line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

The colon (after pos) may be omitted if it would be the last character on the line.

Example

```
100 DELETE #1, REC=8:
```

and

```
100 DELETE #1, REC=8
```

are equivalent.

Description

The DELETE statement specifies that the record indicated by the KEY or RECORD clause is to be deleted from a keyed or relative file. After deletion, the file pointer is positioned immediately after the deleted record.

ERROR CONDITIONS: Three error conditions may be recoverable if an err clause for the condition is specified in the statement or on the referenced EXIT statement:

1. The NOKEY condition occurs if a specified key does not exist on a keyed file.

2. The NOREC condition occurs if a specified record does not exist on a relative file

3. The IOERR condition occurs if a hardware malfunction or other condition prevents deletion of the record.

   The file must be opened with access OUTIN.

The error conditions interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 DELETE #5, RECORD=12: NOREC 200
```

The record with relative record position 12 is to be deleted from file #5. If no such record exists, control is transferred to line number 200.

DIM STATEMENT

The DIM statement specifies the size of arrays and the length of character variables and array elements.

Format

```
DIM m1[,m2]...
```

Where:

Each m may be one of the following:

a

b[*L]

and:

a
is a numeric array declarator.

b
is a character variable or character array declarator.

L
is an unsigned integer constant between one and the maximum string size.

array declarator
has the form:

```
C(d1[,d2[,d3[,d4[,d5[,d6[,d7]]]]]])
```

where:

- Each C is a numeric or character identifier
- Each d is an unsigned integer constant in the range 0 or 1 (depending upon OPTION BASE) to 32767.

Description

The DIM statement explicitly defines the number and extents of array dimensions and the maximum string length of character variables and array elements.

Arrays may be defined with from one to seven dimensions, and each dimension may have a value in the range 0 to 32767 if OPTION BASE 0 is in effect, or 1 to 32767 if OPTION BASE 1 is in effect.

The length of a character variable or array as specified in a DIM statement is the maximum length which that variable or array element may assume within the program unit. If no length is specified in a DIM statement, a character variable or array element has a default maximum length determined by your system administrator. (The IBM-supplied default is 18.)

DIM statements may be placed anywhere in a program unit. They need not appear before use of the arrays and variables they define. Variables and arrays defined in DIM statements may not also appear in COMMON statements.

Example

```
100 DIM A$*72,B$(3,3),C$(200)*5
110 DIM X%(100)
```

A$ is a character variable with a maximum length of 72.

B$ is a 4-by-4 character array with each element having a maximum default length.

C$ is a character array having 201 elements, each with a maximum length of 5 (OPTION BASE 0 is in effect.)

X% is an integer array with 101 elements.

Immediate Execution

DIM can be used in immediate mode to establish the dimensions and maximum string lengths of immediate arrays and character variables. The format and description of an immediate DIM statement are the same as for a DIM statement in a program.

See "Immediate Statements" on page 260 and "Variables and Arrays and Immediate Statements" on page 261 for the rules regarding the interaction with other immediate statements and program statements.

DO Statement

DO STATEMENT

The DO statement initiates the execution of a set of statements that may be processed zero or more times.

Format

```
DO  [{UNTIL|WHILE} logical expression]
```

Where:

logical expression
can be any logical expression as documented in "Logical Expressions" on page 31.

Description

The DO statement is used in conjunction with the LOOP statement to define a loop.

If execution of a program reaches a DO statement, either as initial entry to the loop or when iterating the loop body, the next statement is executed if there is no WHILE or UNTIL clause. If either of these clauses are present, the logical expression is evaluated.

For a WHILE clause, if the expression is true, the next statement is executed, if the expression is false, the statement immediately following the associated LOOP statement is executed (the loop is skipped).

For an UNTIL clause, if the expression is false the next statement is executed, if the expression is true, the statement immediately following the LOOP statement is executed (the loop is skipped).

The values in the expression associated with a DO statement can be set outside the loop and changed within the loop. The expression is reevaluated each time the loop is entered or processed. See "Loop Control Statements" on page 62, "LOOP Statement" on page 177, and "EXIT IF Statement" on page 124.

Example

```
100 LET INC = 9.0
120 DO UNTIL INC = 27.0
130    LET SQYD = 12.0*INC/9.0
140    PRINT SQYD,INC
150    LET INC= INC+1.0
160 LOOP
170 A = B+C
```

In this example, statement 100 sets the initial value of INC to 9.0. The DO clause is evaluated at 120, and it specifies that the statements within the DO loop (130 through 150) are executed until the value in INC equals 27.0. When INC equals 27.0, statement 170 is executed.

ELSE STATEMENT

The ELSE statement specifies the beginning of the ELSE block portion of an IF block.

Format

```
ELSE
```

Description

The ELSE statement is an optional part of an IF block.

The ELSE statement is followed by a group of statements referred to as an ELSE block which are executed if the logical expression in an IF line is false.

The ELSE block is terminated by the END IF statement.

The ELSE statement is also discussed under "IF Blocks" on page 64 and "Block IF Statement" on page 150.

END Statement

END STATEMENT

The END statement indicates both the physical and logical end of the main program.

Format

```
END [numeric expression]
```

Where:

numeric expression
can be any numeric expression

Description

When an END statement is encountered, all open files are closed and the current program is ended.

The optional expression may be any numeric expression. Its purpose is to return a value to the operating environment when the program finishes running in the batch environment. The rounded integer value of the expression is returned.

In the interactive environment, the value of numeric expression is displayed as part of the ending message.

If the main program is missing an END statement and the end of the workspace is encountered, an error message is given and the END statement is assumed. The END statement is also assumed if a SUB statement is encountered during the processing of the main program.

END IF STATEMENT

The END IF statement signifies the end of an IF block.

Format

```
END IF
```

Description

The statements following END IF are executed after the associated THEN block or ELSE block (if any) is executed.

The END IF statement is also discussed under "IF Blocks" on page 64 and "Block IF Statement" on page 150.

END SELECT Statement

END SELECT STATEMENT

The END SELECT statement signifies the end of a SELECT block.

Format

```
END SELECT
```

Description

The END SELECT statement is used with the SELECT, CASE, and CASE ELSE statements to terminate a SELECT block.

The END SELECT statement is also discussed under "SELECT Blocks" on page 66 and "SELECT Statement" on page 252.

END SUB STATEMENT

The END SUB statement marks the physical end of a subprogram.

Format

```
END SUB
```

Description

If an END SUB statement is processed, it acts as a SUBEXIT statement and stops the subprogram, returning control to the caller.

The END SUB statement is also discussed under "Subprogram Statements" on page 78 and "SUB Statement" on page 254.

EXIT STATEMENT

The EXIT statement specifies where control is to be transferred if a particular condition occurs during the execution of an input/output statement.

Format

```
EXIT condition line-ref [,condition line-ref] ...
```

Where:

condition
is CONV, DUPKEY, DUPREC, ENDPAGE, EOF, IOERR, NOKEY, NOREC, or SOFLOW. A single condition may not appear more than once.

line-ref
is a line number or line label.

Description

The EXIT statement is a nonexecutable statement used in conjunction with input/output statements. The EXIT statement specifies a line number or line label to which control is transferred, if an error condition of the type specified occurs in the input/output statement referring to the EXIT statement.

EXIT statements interact with ON condition statements as described in "Exception Handling in I/O Statements" on page 84.

Using an input/output statement with error clauses other than EXIT is equivalent to using the statement with an EXIT error clause and its corresponding EXIT statement.

Example

```
100 GET #5 : A$ EOF 500,IOERR 600,&
  &             CONV 700,SOFLOW 800

or

200 GET #5 : A$ EXIT 300
300 EXIT EOF 500,IOERR 600,CONV 700,&
  &             SOFLOW 800
```

In the above example, the two GET statements are functionally equivalent. During execution of both GET statements, if an error condition occurs, control is passed to the same locations.

The following list shows conditions for which tests may be made:

Condition	Description
CONV	The field cannot be converted to the type of variable specified. An attempt is made to write numeric data using either a C or V FORM specification. A FORM/IMAGE specification refers to a location outside the record. A data list value cannot be converted to the format defined in an associated FORM statement.

EXIT STATEMENT

The EXIT statement specifies where control is to be transferred if a particular condition occurs during the execution of an input/output statement.

Format

```
EXIT condition line-ref [,condition line-ref] ...
```

Where:

condition
is CONV, DUPKEY, DUPREC, ENDPAGE, EOF, IOERR, NOKEY, NOREC, or SOFLOW. A single condition may not appear more than once.

line-ref
is a line number or line label.

Description

The EXIT statement is a nonexecutable statement used in conjunction with input/output statements. The EXIT statement specifies a line number or line label to which control is transferred, if an error condition of the type specified occurs in the input/output statement referring to the EXIT statement.

EXIT statements interact with ON condition statements as described in "Exception Handling in I/O Statements" on page 84.

Using an input/output statement with error clauses other than EXIT is equivalent to using the statement with an EXIT error clause and its corresponding EXIT statement.

Example

```
100 GET #5 : A$ EOF 500,IOERR 600,&
  &             CONV 700,SOFLOW 800

or

200 GET #5 : A$ EXIT 300
300 EXIT EOF 500,IOERR 600,CONV 700,&
  &             SOFLOW 800
```

In the above example, the two GET statements are functionally equivalent. During execution of both GET statements, if an error condition occurs, control is passed to the same locations.

The following list shows conditions for which tests may be made:

Condition	Description
CONV	The field cannot be converted to the type of variable specified. An attempt is made to write numeric data using either a C or V FORM specification. A FORM/IMAGE specification refers to a location outside the record. A data list value cannot be converted to the format defined in an associated FORM statement.

There are not enough values in the record for the data list items.

There is not enough room in the record to write all of the data list items, and SKIP REST is not specified.

Note: The previous three CONV conditions are for record-oriented nonstream input/output only.

DUPKEY A record already exists on the referenced file with the same key as the one specified for the current record.

DUPREC A record already exists on the referenced file with the same record number as the one specified for the current record.

ENDPAGE A PRINT or PRINT File statement has attempted to start a new line beyond the limits specified for the current page.

(A PRINT or PRINT File statement prints as many lines on a page as specified by the BOTTOM parameter in a MARGIN or MARGIN File statement (or, for a PRINT File statement, a default number of lines).)

See also "MARGIN Statement" on page 178 and "ON Condition Statement" on page 203.

EOF There is no more data in a stream-oriented file to satisfy an INPUT or GET statement.

There are no more records in a record-oriented file to satisfy an INPUT, LINE INPUT, or READ statement.

There is insufficient room in a file to accommodate a PUT, WRITE, PRINT, or REWRITE statement.

A CLOSE statement cannot be completed because of lack of space.

IOERR A hardware malfunction prevents record access and could prevent recognition of other exception conditions.

Any input/output error not covered by one of the other input/output conditions.

A format item other than C, V, NC, or PIC in a FORM statement is referred to by a PRINT or PRINT File statement.

NOKEY No record exists in the referenced file with the key specified.

NOREC No record exists in the referenced file with the record number specified.

PAGEOFLOW (See ENDPAGE.)

SOFLOW There are not enough characters in the receiving variable or image to contain the data received, or not enough characters in the definition of the output item to contain all of the characters specified by the output list-item.

Construction of a character string exceeds the maximum allowed.

For a further discussion of the EXIT statement and its relationship to program exceptions, see "Exception Handling Statements" on page 84.

EXIT IF STATEMENT

The EXIT IF statement can be used within a DO or FOR loop to transfer control to the statement immediately following the associated LOOP or NEXT statement when the EXIT IF clause is true.

Format

```
EXIT IF logical expression
```

Where:

logical expression
can be any logical expression as described in "Logical Expressions" on page 31.

Description

The EXIT IF statement may appear within either a DO or FOR loop.

If an EXIT IF statement is reached during normal processing, the logical expression is evaluated:

- If it is false, processing proceeds sequentially.
- If it is true, execution branches to the statement immediately following the innermost loop in which the EXIT IF occurs.

Example 1

```
100 DO WHILE...
.
.
.
150 EXIT IF A=0
.
.
.
180 LOOP
190 PRINT...
```

At line 150, the program goes to line 190 if A is equal to 0. As long as A is not equal to 0 and the WHILE condition is true, lines 100-180 are executed.

Example 2

```
100 FOR A = ...
120 FOR B= ...
.
.
.
190 EXIT IF X=Y
200 FOR C= ...
.
.
.
230 NEXT C
.
.
.
260 NEXT B
265 PRINT X,Y,A,B
270 NEXT A
```

At line 190 (within the FOR/NEXT loop B) the program goes to statement 265 if X is equal to Y. As long as X is not equal to Y, loop C (lines 200-230) is executed for each execution of loop B (lines 120-260).

When X is found equal to Y, processing bypasses loop C and processes the first statement after the NEXT statement of loop B. (In this case, PRINT statement line 265.)

FNEND Statement

FNEND STATEMENT

The FNEND statement indicates both the physical and logical end of a multiline user-defined function.

Format

```
FNEND
```

Description

The FNEND statement marks the physical and logical end of a multiline user-defined function. To exit from a multiline function, the FNEND statement must be executed.

The FNEND statement must be preceded by a DEF statement.

See "User-Defined Function Statements" on page 77 and "DEF Statement" on page 109.

FOR STATEMENT

The FOR statement initiates a FOR/NEXT count-condition loop.

Format

```
FOR var=expression1 TO expression2  [STEP expression3]
```

Where:

var
is a numeric variable.

expression1, expression2, expression3
are numeric expressions.

Description

The FOR and NEXT statements form a FOR/NEXT count-condition loop. The FOR statement is the first statement in the loop; the NEXT statement is the last statement in the loop.

The FOR statement must be matched with a NEXT statement.

The FOR and NEXT statements are paired, with the same controlling numeric variable occurring in both statements. The NEXT statement must follow the paired FOR statement in line number sequence.

The three numeric expressions are evaluated only during the initial processing of the FOR statement including, if necessary, conversion to the type of the control variable according to the rules for numeric conversion. The three expressions are not affected by any statement within the FOR loop.

The numeric variable, var, is the control variable and is modified within the FOR loop, as follows:

1. When the loop is first processed, the control variable is set to the initial value, expression1.

2. If expression1, the initial value, is greater than (or, for negative increments, less than) the expression following the TO (expression2) at evaluation time, the loop is never processed and the value of the control variable is set to the initial value (exp1).

3. If expression1, the initial value, is less than or equal to expression2, the expression following TO, the statements in the loop are processed, and the expression following STEP (expression3) is added to the control variable.

   If STEP expression3 is omitted, the increment is automatically set to 1.

   (The optional STEP parameter, STEP expression3, bypasses unnecessary values by supplying an increment other than 1.)

4. This process continues until the control variable is greater than (or, for negative increments, less than) the expression following TO (expression2).

5. Control now passes to the first statement following the NEXT statement.

It is possible to transfer control out of a FOR/NEXT loop. In this case, the control variable retains its value at the time of the

transfer until either reset outside of the loop or reset by reentry through the FOR statement.

Except for the CONTINUE, RETRY, and RETURN statements, control cannot enter a loop unless it enters at the initial FOR statement.

FOR/NEXT loops are also described in "Loop Control Statements" on page 62.

Example

```
110 FOR FEET=9.0 TO 48.0 STEP 3
120     LET YARDS=12.0*FEET/9.0
130     PRINT YARDS,FEET
140 NEXT FEET
150 END
```

This loop is executed fourteen times. During the first iteration, FEET is 9, during the second iteration, FEET is 12, etc. After the last iteration, when another iteration would increase FEET beyond 48, the loop is exited.

FORM STATEMENT

The FORM statement defines the exact appearance of both input and output data.

Format

```
FORM item [,item]...
```

Where:

item
can be literal, or control specification, or [repeat*]data form.

And where:

literal
is a quoted character string.

control specification
is one of the following:

X[e]
skip e positions in record or on line

POS[e]
position to location e in record or on line

SKIP[e]
skip e number of lines

Where:

e
is a numeric expression evaluating to a rounded integer.

[NEW]PAGE
position to top of new page

repeat
is an unsigned, nonzero integer constant or variable, used as a replication factor with a data form.

data form
is one of the data forms shown in Figure 24 on page 130.

Data Form	Meaning
PIC(s[s]...[¬¬¬[¬...]][tr])	Picture of data item
C[w]	Character data
V[w]	Character data with trailing blanks removed on input
N w[.d]	Conversion of numeric data to and from character data
G[w[.d]]	Represents either character data or conversion of numeric data to and from character data, depending upon the type of the data, character or numeric
NC w[.d]	Conversion of numeric data to zoned decimal format on output, and conversion of either zoned decimal or numeric characters on input
ZD w[.d]	Conversion of zoned decimal data for both input and output
B[w]	Fixed-point binary
S	Short-form floating-point binary (32 bits)
L	Long-form floating-point binary (64 bits)
PD w[.d]	Packed decimal
ND	Internal floating-point decimal
NI	Internal integer

Where:

w
is an unsigned, nonzero integer constant, which may be preceded with blanks.

d
is an unsigned, integer constant.

s
is a digit specifier (#, Z, *, $, +, or -), or an insertion character (a comma (,), solidus (/), blank (B), or decimal point (.).

¬
is an exponent specifier, where three or more (¬) characters are shown. (Can also be specified as the circumflex character.)

tr
is a trailing character, that is, a trailing plus (+), trailing minus (-), trailing credit (CR), or either form of trailing debit (DB or DR).

Figure 24. FORM Statement Data Form Codes

Description

The FORM statement is used in conjunction with the PRINT, PRINT file, READ, REREAD, WRITE, and REWRITE statements. These statements may reference the line number or line label of a FORM statement, or a character expression which evaluates to a FORM statement.

The FORM can specify literal values, the format of character and numeric data (data form specifications), and the positioning of data (control specifications), all of which describe the components of a record or line of data.

Each data item of the input or output list of the input/output statement is matched against a corresponding data form specification in the FORM statement. If there are more list items than data form specifications, the FORM is reused from its beginning until the list is exhausted; an excess of data form specifications over list items is ignored.

Literal Specifications

If a quoted character string appears in a FORM associated with a READ or REREAD statement, it is treated as if it were an X[n] control specification, where the value of the n is equal to the number of characters in the character string, excluding the surrounding quotation marks. The effect is the skipping of n positions of the input record.

Example

```
110 FORM "INPUT",N5.2,C5
120 READ #5 USING 110: NUM#,CHAR$
```

when the READ file statement is executed, 5 characters of the input data are skipped, and data transfer begins with NUM#.

If a quoted character string appears in a FORM associated with a PRINT, PRINT File, WRITE, or REWRITE statement, it is treated as if it were a C[w] data form specification, where w is equal to the number of characters in the character string (excluding the surrounding quotes). The effect is the transmission of w characters to the output record.

Example

```
110 FORM "OUTPUT",N5.2,C5
120 WRITE #5 USING 110: NUM#,CHAR$
```

when the WRITE file statement is executed, the characters OUTPUT are sent to the output device, followed by the contents of NUM# and CHAR$.

Control Specifications

Control specifications set the position within a record or line (POS and X), and control line skipping (SKIP and PAGE).

The control specifications X, POS, and SKIP are optionally followed by a numeric parameter. When these control specifications are used in a FORM statement, the parameter may be any numeric expression. However, when an input/output statement uses a character expression as a FORM (the USING clause refers to a character expression rather than a FORM statement), the parameter for all control specifications within the character string must be numeric constants.

X The X [e] control specification indicates how many positions are to be passed over in the line or record up to the next value.

If X is specified in a FORM statement, it must be followed by a space; however, if a FORM is contained in a character expression, the space after the X may be omitted.

e is a numeric expression evaluating to a rounded expression greater than zero. It may not refer to user-defined functions.

For READ, REREAD, WRITE, and REWRITE operations, the value of e must be between 1 and the defined record length. If e is less than 1 or is omitted, 1 is assumed; if e is greater than the record length, a CONV error occurs.

For PRINT and PRINT File operations, if the resultant position is greater than the right margin value, the current line or record is assumed to be complete and is transmitted to the output device or file, resetting the position to the left margin of the next line or record.

Example

```
300 PRINT #3 USING 400 : A$,B$
400 FORM C10,X 5,C10
```

The value of A$ will start 9 characters to the right of the left margin. The value of B$ will start 15 characters to the right of the left margin.

POS The POS [e] control specification indicates the position in the record or line for the next value.

e is a numeric expression evaluating to a rounded expression greater than zero. It may not refer to user-defined function.

If POS is specified in a FORM statement, it must be followed by a space; however, if a FORM is contained in a character expression, the space after the POS may be omitted.

For READ, REREAD, WRITE, and REWRITE operations, the value of e must be between 1 and the defined record length. If e is less than 1 or is omitted, 1 is assumed; if e is greater than the record length, a CONV error occurs.

For PRINT and PRINT File operations, the value of e must be between the left and right margin values (see the "MARGIN Statement" on page 178). If e is less than the left margin value, the left margin value is assumed; if e is greater than the right margin value, the current line or record is assumed to be complete and is transmitted to the output device or file, resetting the position to the left margin of the next line or record.

Example

```
100 WRITE #10 USING 200 : A$,B%
200 FORM C15,POS 24,N6
```

The value of A$ will be written in positions 1-15; B% will be written in positions 24-29.

SKIP The SKIP [e] control specification indicates how many lines are to be skipped before the next value is printed; this option is valid only with the PRINT or PRINT File statement. The numeric expression (e) may not refer to user-defined functions.

If SKIP is specified in a FORM statement, it must be followed by a space; however, if a FORM is contained in a character expression, the space after the SKIP may be omitted.

If e is less than 0 or is omitted, the value 1 is assumed.

If e is 0, the action taken depends upon the file being printed:

- If the file contains a carriage control character at the beginning of each record (that is, the file is opened as DEVICE PRINTER or DEVICE 3800, see "OPEN Statement" on page 206), the current image of the

record is written with no line advance. The next data to be printed begins a new record which prints over the previous record.

- If the file does not have carriage control characters or if the option is to PRINT to the terminal, SKIP 0 is handled the same as POS 1; that is, characters are overlaid in the current record but the effect is replacement rather than overprinting.

When e is greater than 0, a current image of the line or record is created, and e-1 blank lines or records are generated. The position of the next line or record is set to the left margin value.

If e is greater than the remaining lines on the page, then e is taken as the number of lines remaining on the page. This causes blank lines to be printed until an ENDPAGE condition is generated (see "MARGIN Statement" on page 178 and "ON Condition Statement" on page 203).

Example

```
100 PRINT USING 200 : A$,B%
200 FORM C15,SKIP 7,N3
```

Six blank lines will appear between the value of A$ and the value of B%.

PAGE The PAGE control specification indicates that the next value is to be written on a new page positioned to the left margin; this option is only valid in conjunction with PRINT or PRINT File statements.

If PAGE or NEWPAGE is specified and the file is not opened as DEVICE PRINTER or DEVICE 3800 (see "OPEN Statement" on page 206), an exception is generated. The SYSTEM action for the exception is a warning message.

When used with a PRINT statement to a display terminal, PAGE clears the screen.

PAGE also resets the internal line counter used to control the top and bottom margins, see "MARGIN Statement" on page 178.

Example

```
100 PRINT #5 USING 200 : A$,B%
200 FORM C10,PAGE,N5
```

The values of B% will appear at the top of the page following the page on which A$ appears.

Data Form Specifications

Data form specifications indicate the formats in which character and numeric input data items exist, and the formats in which character and numeric output data items are to be created.

Data form specifications may be preceded by a repeat count, or replication factor. Within a FORM statement, the replication factor can be either an integer constant or an integer variable. But, within FORM specifications that are character strings specified in the USING clause of input/output statements, the replications factors must be integer constants only.

Only the C[w], V[w], Nw[.d], and PIC data form specifications can be used in FORM statements associated with PRINT and PRINT File statements.

PIC

The PIC data form specification may only be used on FORM statements associated with the output statements PRINT, PRINT File, WRITE, and REWRITE.

PIC identifies the placement of a character or numeric output list item in an output record. The keyword PIC is followed by a field of characters that indicate various types of formatting as described below; the field of characters is enclosed in parentheses and is referred to as a picture field.

For character data, the contents of the picture field are ignored and only its length is used. If the data value has the same length as the picture field, the data value is moved to the output record, exactly as it appears. If the data value is shorter than the picture field, the character string is padded with blanks on the right. If the character value is longer than the picture field, a string overflow exception occurs.

For numeric data, the picture field specifies both length and the format for its value. The specification consists of digit specifiers, insertion characters, and exponent specifiers.

The digit specifiers and their functions are as follows ('b' in output implies a space):

Specifies a data position where a digit must always appear.

```
                                  Value        Output
  100 FORM PIC (#####)             63          00063
```

Z Causes zero suppression; that is, a leading zero in the associated data position is replaced by a blank.

```
                                  Value        Output
  110 FORM PIC (ZZZ.##)           24.6         b24.60
```

When zero suppression is in effect, at least one decimal position to the left of the decimal place is printed, unless the entire field is zero. In that case, the zero to the left of the decimal point is omitted.

***** Causes zero suppression; that is, a leading zero in the associated position is replaced by an asterisk.

```
                                  Value        Output
  120 FORM PIC (***##)            243          **243
                                   -6          -**06
```

Note: The use of both Z and * in a single edit string is not valid.

$ Specifies each data position that can potentially be occupied by a floating currency sign (that is, a currency sign immediately to the left of the first significant digit).

If a single $ is used in the PIC, the $ appears in the leftmost position of the field. If more than one $ is used, the $ appears, justified as far to the right as possible, where a $ appeared, and where no significant digit is present.

```
                                  Value        Output
  130 FORM PIC ($$$##)            243          b$243
                                   -6          bb$-6
```

Note: Use of a $ digit specifier has the effect of suppressing nonsignificant leading zeros

+ Specifies each data position where a floating high-order sign may appear. The use of this character guarantees the appearance of either a plus sign or a minus sign in the printed field.

```
                                  Value       Output
  140 FORM PIC (+++##)             243        b+243
                                    -6        bb-06
```

If a single + is used in the PIC, the sign of the numeric value appears in the leftmost position of the field. If more than one + sign is used, the sign of the numeric value appears, justified as far to the right as possible, wherever a plus sign appeared, and where no significant digit is present.

Note: Use of a + digit specifier has the effect of suppressing nonsignificant leading zeros

- Which indicates each data position where a floating high-order minus sign may appear if the field is negative.

```
                                  Value       Output
  150 FORM PIC (---##)             243        bb243
                                    -6        bb-06
```

If a single - is used in the PIC, and the value is negative, the sign of the numeric value appears in the leftmost position of the field. If more than one minus sign is used in the PIC, the sign of the numeric value appears, justified as far to the right as possible, wherever a minus sign appeared, and where no significant digit is present.

Note: Use of a - digit specifier has the effect of suppressing nonsignificant leading zeros.

If the picture field does not contain at least one sign position (leading or trailing), and if the value of the expression is negative, and if the field is large enough to contain a minus sign, a leading minus sign is printed.

Insertion characters are characters inserted into the output at the position they are specified in the PIC clause. Insertion characters may be either conditional or unconditional.

The unconditional insertion character is:

B the blank, which always appears when specified.

The conditional insertion characters are:

, the comma

/ the solidus or slash

. the decimal point

CR the trailing credit symbol

DB or DR the trailing debit symbol

+ or - the trailing sign symbols

The trailing symbols (CR, DB, DR, trailing +, trailing -) are replaced by either a blank or an asterisk if the conditions for their appearance are not met. Only one

trailing symbol representation is permitted per PIC clause, and then only if there is no leading + or -.

The rules for the appearance of the insertion characters are:

- A comma (,) or a solidus (/) is not allowed as the first character in the PIC clause. (The condition for their appearance can never be met).
- If zero suppression is not in effect, or if a significant digit is found to their left, the comma (,) and solidus (/) always appear.
- If zero suppression is in effect because of a preceding conditional digit specifier (Z, *, $, +, -), then if no significant digit exists to their left, the following characters do not appear:

  comma (,)

  solidus or slash(/)

  trailing signs (+ or -)

  They are replaced, instead, by either a blank or (if the preceding digit specifier is an asterisk) an asterisk.

  In this case, the trailing plus sign always causes the appearance of either a positive or a negative sign, while the other trailing symbols (minus sign, CR symbol, DB or DR symbol), appear only when the resulting value is negative.
- The decimal point (.) always appears in its associated position except in the following circumstance: zero suppression is specified for every digit position (both to the left and right of the decimal point) and the value of the numeric field is zero. In this case, the decimal point is replaced by the appropriate zero suppression character.

An exponent specifier appears in the three or more low-order characters of a PIC string.

Three or more occurrences of ¬ specify the presence in the corresponding print positions of the following sequence:

1. The letter E
2. The exponent sign (+ or -)
3. One or more digits representing the value of the exponent

Zero suppression is effectively turned off by an exponent specifier. A decimal point specified previously will therefore always appear in a field defined with an exponent specifier.

PIC Clause Examples

The following examples show some of the different types of conversion specifications that can be used with the PIC clause. Blanks in the result field are shown here as the lower case "b".

Integer Format:

Value	Specification	Result
10	PIC(####)	0010
10	PIC(ZZZZ)	bb10
10	PIC(****)	**10
10	PIC($$$$)	b$10
-10	PIC($$$$$$$$$)	bbbb$-10
20289	PIC(##/##/##)	02/02/89
-123	PIC(ZZZZDB)	b123DB
123	PIC(++++ZZZZ)	bbb+b123
123	PIC(+ZZZZZZZ)	+bbbb123

Fixed Point Format:

Value	Specification	Result
12.145	PIC(ZZZ.##)	b12.15
1000	PIC(***,***.##)	**1,000.00
-77	PIC(####.###)	-077.000
0000	PIC(***.**)	******

Floating Point Format:

Value	Specification	Result
5	PIC(ZZZ.ZZ¬¬¬¬)	500.00E-02
-255.555	PIC(ZZ.ZZZ¬¬¬)	-2.556E+2

Character String:

Value	Specification	Result
ABC	PIC(#####)	ABCbb
ABC	PIC(ZZ.ZZZ)	ABCbbb
XYZ	PIC(###)	XYZ

C[w] This data form specification deals with w positions of character data.

On input, the next w characters from a record are moved to a corresponding character variable in the input list. If the variable's maximum length, n, is less than w, a string overflow occurs. If n is greater than w, the variable's length becomes the length of the character string transmitted. The value of w defaults to 1.

On output, the next w characters in an output record will be the result of the evaluation of a character expression in the output list. If the length (n) of the expression is less than w, then w minus n blanks are added to the right of the expression's value. If n is greater than w, a string overflow occurs. The value of w defaults to 1.

This data form specification is valid for READ, REREAD, PRINT, PRINT File, WRITE, and REWRITE statements.

V[w] On input, the next w characters from the record are inspected and all characters (up to and including the last nonblank character in the field) are moved to a corresponding character variable in an input list (that is, trailing blanks are removed).

If the variable's maximum length, n, is less than the number of characters moved, p, a string overflow occurs.

If n is greater than p, the variable's length becomes p.

The value of w defaults to 1.

For output, the V specification acts exactly as the C specification.

The data form specification is valid for READ, REREAD, PRINT, PRINT File, WRITE, and REWRITE statements.

N w[.d] On input, the specified number of positions of the record contain a numeric value in character form which is to be moved to a corresponding numeric variable in the input list. The value is converted to IBM BASIC internal integer or decimal format (with rounding for integer) as required by the type of the receiving variable.

In the record, the numeric data must be a right-justified character string consisting of numeric data in either integer or fixed-point format. If the character string is all blanks, the number is set to 0. The optional constant d indicates the number of decimal fraction positions in the field if the field has no explicit decimal point (an explicit decimal point overrides the d specification).

On output, the corresponding numeric expression is rounded to the number of decimal places specified by d and converted to a character value having a decimal point. This value is placed in the next w positions in the record and right-justified. If d is not specified, d is assumed to be 0 and no decimal point is placed in the field. If the value is negative, a minus sign precedes the value.

The value of d must be less than or equal to w. For example, if a number is negative and contains a decimal point, d must be at least two less than w to allow for the minus sign and the decimal point.

This data form specification is valid for READ, REREAD, PRINT, PRINT File, WRITE, and REWRITE statements.

G[w[.d]] This data form specification can be used to transmit either numeric or character values. It acts as a V data form specification if the value is character, and as an N data form specification if the value is numeric. The value of w defaults to 1.

For input, if the corresponding input list variable is character, the G specification is treated the same as a V specification where the decimal fraction "d" is ignored if specified. If the corresponding input list variable is numeric, the G specification is treated the same as an N specification.

For output, if the corresponding output list item is a character item, the G specification is treated the same as a V specification where the decimal fraction "d" is ignored if specified. If the corresponding output list item is numeric, the G specification is treated the same as an N specification.

This data form specification is valid for READ, REREAD, PRINT, PRINT File, WRITE, and REWRITE statements.

Data Conversion Examples

The following are examples of the C, V and N conversion specifications. The lower case "b" shows blanks in the result.

Output:

Value	Specification	Result (Characters)
AB	C3	ABb
AB	V4	ABbb
AB	C	string overflow error
75	N3	b75
3.45	N7.2	bbb3.45
3.45	N7.1	bbbb3.5
-3.45	N7	bbbbb-3

Input:

Value	Specification	Result (Characters)
AB	C3	ABb
AB	V4	ABbb
AB	C	A
b75	N3	75
bbb3.45	N7.2	3.45
bbbbb-3	N7	-3

NC w[.d] On input, indicates that the next w positions of a record contain a numeric value in zoned decimal format (a zone and a digit per position, except for the low-order position which may contain either a zone or a sign and a digit), or the value which was written by a PIC data form specification. The value is converted to IBM BASIC internal integer or decimal format (with rounding for integer) as required by the type of the receiving variable.

The optional d indicates the number of decimal positions in the field and, if present, will override an explicit decimal point in the data. In addition to digits, of which the rightmost may be signed, the input field may contain a combination of any of the following characters:

$, +, -, *, /, CR, DB, DR, blank, comma (,), decimal point (.), exponential notation (E plus or minus numeric constant)

On output, indicates that the corresponding numeric expression in an output list is to be converted to a signed zoned decimal field of length w, rounded, and placed in the output record.

If the optional parameter d is present, that number of decimal positions will be present in this field; if it is not, all w positions will represent the integer part of the numeric value. The value of d must be less than or equal to the value of w.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

ZD w[.d] On input, the next w positions of the record contain a numeric value in zoned decimal form which is to be converted to internal numeric representation and moved to a corresponding numeric variable in the input list.

The optional parameter, d, specifies the number of digits in the fractional portion of the number, and is assumed to be zero if absent.

On output, the corresponding numeric expression in an output list is to be converted to a signed, zoned, decimal field of length w, rounded, and placed in the output record.

If the optional parameter, d, is present, d decimal positions will be present in this field; if not, all w positions will represent the integer part of the numeric value. The value of d must be less than or equal to the value of w.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

Numeric Data Conversion Examples

The following are examples of the NC and ZD conversion specification.

Input:

Value (Characters)	Specification	Result
bb$1,234.56CR	NC13.3	-123.456
bb$1,234.56CR	ZD13.3	error
000034E	NC7.2	3.45
000000L	NC7	-3
000000L	ZD7.1	-.3

Output:

Value (decimal)	Specification	Result (hexadecimal)
3.45	NC7.2	F0F0F0F0F3F4C5
3.45	ZD7.2	F0F0F0F0F3F4C5
-3.45	NC7	F0F0F0F0F0F0D3

Note: Hexadecimal values shown are the EBCDIC character equivalent of the numbers used. See "Appendix B. Character Set Collating Sequences" on page 327 for the characters these codes represent.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

B [w] For input, the next w positions (w must be 2, 4, or 8) contain the binary representation of a numeric value. This value is to be assigned to the corresponding numeric variable in the input list. The default value for w is 4.

For output, the corresponding numeric expression in an output list is converted to a rounded, fixed-point binary integer, occupying the next w record positions. w must be 2, 4, or 8. The default is 4.

Example

```
100 I=10
110 J=14
120 WRITE #2 USING 130:I,J
130 FORM B4,B4
```

In this example I and J will be written to the file and the binary value 0...01010 will occupy the first four bytes of the record and the binary value 0...01110 will occupy the next four bytes of the record.

On output, the corresponding numeric expression in an output list is to be converted to a signed, zoned, decimal field of length w, rounded, and placed in the output record.

If the optional parameter, d, is present, d decimal positions will be present in this field; if not, all w positions will represent the integer part of the numeric value. The value of d must be less than or equal to the value of w.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

Numeric Data Conversion Examples

The following are examples of the NC and ZD conversion specification.

Input:

Value (Characters)	Specification	Result
bb$1,234.56CR	NC13.3	-123.456
bb$1,234.56CR	ZD13.3	error
000034E	NC7.2	3.45
000000L	NC7	-3
000000L	ZD7.1	-.3

Output:

Value (decimal)	Specification	Result (hexadecimal)
3.45	NC7.2	F0F0F0F0F3F4C5
3.45	ZD7.2	F0F0F0F0F3F4C5
-3.45	NC7	F0F0F0F0F0F0D3

Note: Hexadecimal values shown are the EBCDIC character equivalent of the numbers used. See "Appendix B. Character Set Collating Sequences" on page 327 for the characters these codes represent.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

B [w] For input, the next w positions (w must be 2, 4, or 8) contain the binary representation of a numeric value. This value is to be assigned to the corresponding numeric variable in the input list. The default value for w is 4.

For output, the corresponding numeric expression in an output list is converted to a rounded, fixed-point binary integer, occupying the next w record positions. w must be 2, 4, or 8. The default is 4.

Example

```
100 I=10
110 J=14
120 WRITE #2 USING 130:I,J
130 FORM B4,B4
```

In this example I and J will be written to the file and the binary value 0...01010 will occupy the first four bytes of the record and the binary value 0...01110 will occupy the next four bytes of the record.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

S On input, indicates that the short form of a floating-point operand (32 bits) exists in the record, and is to be moved to the corresponding variable in the input list. The value is converted to decimal or integer format as required by the receiving variable.

On output, specifies that a numeric value is to be converted to the short form of a floating-point operand and written to the record.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

L On input indicates that the long form of a floating-point operand (64 bits) exists in the record, and is moved to a corresponding numeric variable in the input list. Conversion to decimal or integer format is performed as required by the receiving variable.

On output, specifies that a numeric value is to be converted to the long form of a floating-point operand and written to the record.

Floating-Point Conversion Example

```
100 I#,J#=3.45
110 WRITE #3,USING 120:I#,J#
120 FORM S,L
```

In this example, I# and J# will be written to the file. The first four bytes will contain the short precision numeric values of I# and the next eight bytes will contain the long precision numeric value of J#.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

PD w[.d] On input, the next w positions of the record contain a numeric value in packed decimal form (two digits per position, except for the low-order position holding one digit and a sign), which is to be converted to internal numeric representation and moved to a corresponding numeric variable in the input list. The optional parameter, d, specifies the number of digits in the fractional portion of the number, and, if absent, is assumed to be zero.

On output, the corresponding numeric expression in an output list is to be converted to a packed decimal field with d fractional digits, occupying the next w record positions.

The value of d must be less than or equal to the value of w.

Packed Decimal Conversion Example

Value	Specification	Result (hexadecimal)
3.45	PD 7.2	0000000000345C
3.37	PD 7.1	000000000034C
-3.29	PD 7	0000000000003D

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

ND On input, the internal representation of a decimal value exists in the record (12 character positions) and is to

be moved to a corresponding numeric variable in the input list. If the variable has integer type, the value is converted to integer format with rounding.

On output, a value is written to the next 12 character positions in the record in internal floating decimal format. Integer values are converted to their decimal equivalent.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

NI On input, the internal representation of an integer value exists in the next 4 character positions in the record and is to be moved to a corresponding numeric variable in the input list. If the variable has decimal type, the value is converted to decimal format.

On output, a numeric value is written to the record in the next 4 character positions in internal integer format. Decimal values are rounded and converted to their integer equivalent.

This data form specification is valid for READ, REREAD, WRITE, and REWRITE statements.

Internal Integer Conversion Example

```
100 J# = 15.36
200 I# = 124.3
300 WRITE #3 USING 400 : I#,J#
400 FORM ND,NI
```

This sequence of code causes the IBM BASIC internal representation of the decimal number 124.3 (12 bytes) to be moved to the output record. The internal representation of the decimal number 15.36 is then rounded to an integer, converted to a 4-byte binary value, and moved to the next four bytes in the output record.

GET STATEMENT

The GET statement retrieves values from a stream or internal file.

Format

```
[MAT] GET #fileref : input-list [,SKIP REST]
          [err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, is within the range 1 to 255. It identifies the file to be processed.

input-list
is an input list of variable or array names (possibly subscripted) separated by commas.

err
is one of the following:

EXIT line-ref

EOF line-ref

IOERR line-ref

CONV line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

For stream files, the GET statement retrieves one data value at a time and assigns it to the corresponding item of the input list, if necessary, getting additional records to satisfy the input list.

For internal sequentially organized files not open with stream organization, the GET statement retrieves one record and assigns each of its values to the corresponding items of the input list. An internal sequentially organized record file may be opened with stream organization specified. In this case the GET statement acts as if the file was created as a stream organized file.

For both types of files, each value retrieved and assigned must be of the same basic type (character or numeric) as the corresponding variable in the input list or a conversion condition occurs. However, numeric values can be assigned to either integer or decimal variables, with conversions being made as for the LET statement. If an input list item is subscripted, the subscripts are evaluated just before the value is assigned.

MAT KEYWORD: The MAT keyword preceding the GET keyword specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input-list.

If MAT does not precede GET, then any individual array item in the input-list may be preceded by the MAT keyword.

See "Input/Output Lists" on page 70 for more information.

FILEREF: The file reference must refer to a stream or internal file. (See "Combinations of File Organization and Format" on page 57.)

INPUT-LIST: An array in an input list is identified by the keyword MAT appearing before the array name. (Unless the entire GET statement is prefaced with MAT, in which case all list items must be arrays and individual MAT specifications are unnecessary.) Arrays are assigned values from the specified file with the rightmost subscript varying most rapidly. If the array name is followed by redimension specifications, the array is first redimensioned to extents equal to the rounded integer values of the numeric redimension expressions, and then the array is filled. When an array is redimensioned, the original number of members may not be exceeded.

Ordinarily, the length of a receiving character variable is set to the length of the character string assigned to it. However, if the length of the character string exceeds the maximum length of the receiving variable, a string overflow occurs.

If there are no more values in the file to assign to remaining input list items, an end-of-file condition exists.

SKIP REST CLAUSE: For an internal sequential file, a conversion error occurs if all the items in the input list have been satisfied, but more values exist in the current record; this situation can be avoided by use of the SKIP REST clause which indicates that all remaining values in a record are to be ignored.

Because stream files have only one value per record, the SKIP REST clause has no meaning and is ignored if it appears on the GET statement.

ERROR CONDITIONS: The string overflow (SOFLOW), conversion (CONV), and end-of-file (EOF) conditions described above, as well as the input/output error condition (IOERR), may be recoverable if the corresponding error clauses are included on the GET statement. For example, an attempt to get from fileref 0, or an attempt to get from a file opened for OUTPUT, are situations which result in an IOERR condition.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 GET #27 : A%,B$
200 GET #27 : X%,MAT Z%(X%)
```

The first GET statement results in values being assigned to A% and B$. The second GET statement assigns a value to X%, redimensions the array Z%, and assign values to the newly-dimensioned array.

If MAT does not precede GET, then any individual array item in the input-list may be preceded by the MAT keyword.

See "Input/Output Lists" on page 70 for more information.

FILEREF: The file reference must refer to a stream or internal file. (See "Combinations of File Organization and Format" on page 57.)

INPUT-LIST: An array in an input list is identified by the keyword MAT appearing before the array name. (Unless the entire GET statement is prefaced with MAT, in which case all list items must be arrays and individual MAT specifications are unnecessary.) Arrays are assigned values from the specified file with the rightmost subscript varying most rapidly. If the array name is followed by redimension specifications, the array is first redimensioned to extents equal to the rounded integer values of the numeric redimension expressions, and then the array is filled. When an array is redimensioned, the original number of members may not be exceeded.

Ordinarily, the length of a receiving character variable is set to the length of the character string assigned to it. However, if the length of the character string exceeds the maximum length of the receiving variable, a string overflow occurs.

If there are no more values in the file to assign to remaining input list items, an end-of-file condition exists.

SKIP REST CLAUSE: For an internal sequential file, a conversion error occurs if all the items in the input list have been satisfied, but more values exist in the current record; this situation can be avoided by use of the SKIP REST clause which indicates that all remaining values in a record are to be ignored.

Because stream files have only one value per record, the SKIP REST clause has no meaning and is ignored if it appears on the GET statement.

ERROR CONDITIONS: The string overflow (SOFLOW), conversion (CONV), and end-of-file (EOF) conditions described above, as well as the input/output error condition (IOERR), may be recoverable if the corresponding error clauses are included on the GET statement. For example, an attempt to get from fileref 0, or an attempt to get from a file opened for OUTPUT, are situations which result in an IOERR condition.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 GET #27 : A%,B$
200 GET #27 : X%,MAT Z%(X%)
```

The first GET statement results in values being assigned to A% and B$. The second GET statement assigns a value to X%, redimensions the array Z%, and assign values to the newly-dimensioned array.

GOSUB STATEMENT

The GOSUB statement is used, together with the RETURN statement, to invoke subroutines.

Format

```
GOSUB {line-number|line-label}
```

Where:

line-number
must be an existing line-number in the program.

line-label
must be an existing line-label in the program.

Description

The GOSUB statement may be spelled GO SUB.

The GOSUB statement causes the program to branch to the indicated line number or line label.

The RETURN statement transfers control to the first executable statement following the last GOSUB statement which was executed.

The statements executed between the time a GOSUB statement transfers control and control is returned by a RETURN statement are called a subroutine. The last statement executed in a subroutine is a RETURN statement. Its purpose is to allow normal sequential processing to continue under completion of the subroutine specified by the GOSUB statement.

An attempt to branch via the GOSUB statement to a nonexistent line number or line label results in a warning message when the program is compiled or when a RUN command is issued. When such a GOSUB statement is executed, an exception is generated. This exception can be handled by the ON Condition statement using the ERROR condition. See "ON Condition Statement" on page 203 and "Exception Handling Statements" on page 84.

Processing of a RETURN statement without an active GOSUB statement results in an exception.

More than one GOSUB statement may be active, that is, one subroutine may use the GOSUB statement to branch to another subroutine.

Normally, each GOSUB statement must have a matching RETURN statement. However, it is not necessary to have executed an equal number of GOSUB/RETURN statements when the current program unit is ended during execution. All active GOSUB statements in a subprogram or in a multiline function are set inactive by the execution of either a SUBEXIT statement, or of an FNEND statement.

Example

```
100 GO SUB 140
110 REM
120 REM
130 GO TO 230
140 REM
150 GO SUB A1
160 REM
170 REM
180 RETURN
190 A1: X = X+Y
200 GO SUB 260
210 REM
220 RETURN
230 REM
240 REM
250 GO TO 300
260 REM
270 REM
280 REM
290 RETURN
300 REM
```

The sequence of statement processing from the above program segment would be:

```
100,140,150,190,200,260,270,280,290,210,220
160,170,180,110,120,130,230,240,250,300
```

GOTO STATEMENT

The GO TO statement unconditionally branches to the indicated line number or line label.

Format

```
GOTO {line-number|line-label}
```

Where:

line-number
must be an existing line-number in the program.

line-label
must be an existing line label in the program.

Description

The GOTO statement may be spelled GO TO.

The GOTO statement unconditionally transfers control to the specified line number or line label.

An attempt to go to a nonexistent line number or line label results in a warning message when the program is compiled or when a RUN command is issued. When such a GOTO statement is executed, an exception is generated. This exception can be handled by the ON Condition statement using the ERROR condition. See "ON Condition Statement" on page 203 and "Exception Handling Statements" on page 84.

Example

```
100 GO TO 190
.
.
.
190 LET A = B+C
```

When statement 100 is executed, all statements between 100 and 190 are bypassed. Sequential processing continues from that point.

IF Statement

IF STATEMENT

The IF statement evaluates a logical expression and conditionally transfers control or conditionally executes a statement or series of statements.

Format

```
IF logical expression
           THEN statement-reference
           [ELSE statement-reference]
```

Where:

logical expression
can be any logical expression as described in "Logical Expressions" on page 31.

statement-reference
is a line number, line label, or list of imperative statements separated by end of statement characters (:).

Description

The logical expression in the IF statement is evaluated. If the expression is true, either control is transferred to the line number or line label following THEN or the statements immediately following THEN are executed. Control then passes to the next statement after the IF statement.

Example

```
200 IF A = B THEN 500
210 X = X+Y
```

If the value in A equals the value in B, the statement is true, and control is transferred to line number 500. If A is not equal to B, statement 210 is executed.

If a list of statements follows THEN, and the expression is true, they are processed in sequence.

Example

```
100 IF A=B THEN LET C=D: LET E=F: LET G=H
```

If the logical expression in the IF statement is false, either control is transferred to the line number or line label following ELSE or the statements immediately following ELSE are executed. If there is no ELSE clause, control is transferred to the next statement after the IF statement.

Example

```
200 IF A = B THEN 500 ELSE LET C = E
210 X = X+Y
```

If the value in A is not equal to the value in B, the value of C is set equal to the value of E, because the ELSE clause is executed.

When an IF statement ends with a statement list, all of the rest of the statements on the line are considered part of the list and are executed under control of the IF.

IF Statement

IF STATEMENT

The IF statement evaluates a logical expression and conditionally transfers control or conditionally executes a statement or series of statements.

Format

```
IF logical expression
            THEN statement-reference
            [ELSE statement-reference]
```

Where:

logical expression
can be any logical expression as described in "Logical Expressions" on page 31.

statement-reference
is a line number, line label, or list of imperative statements separated by end of statement characters (:).

Description

The logical expression in the IF statement is evaluated. If the expression is true, either control is transferred to the line number or line label following THEN or the statements immediately following THEN are executed. Control then passes to the next statement after the IF statement.

Example

```
200 IF A = B THEN 500
210 X = X+Y
```

If the value in A equals the value in B, the statement is true, and control is transferred to line number 500. If A is not equal to B, statement 210 is executed.

If a list of statements follows THEN, and the expression is true, they are processed in sequence.

Example

```
100 IF A=B THEN LET C=D: LET E=F: LET G=H
```

If the logical expression in the IF statement is false, either control is transferred to the line number or line label following ELSE or the statements immediately following ELSE are executed. If there is no ELSE clause, control is transferred to the next statement after the IF statement.

Example

```
200 IF A = B THEN 500 ELSE LET C = E
210 X = X+Y
```

If the value in A is not equal to the value in B, the value of C is set equal to the value of E, because the ELSE clause is executed.

When an IF statement ends with a statement list, all of the rest of the statements on the line are considered part of the list and are executed under control of the IF.

Example

```
200 IF A = B THEN GOTO 300
210 X = X + Y
```

is not equivalent to

```
200 IF A = B THEN GOTO 300: X = X + Y
```

In fact, in the second version, X = X + Y will never be executed.

The imperative statements available for use with the IF statement are listed in Figure 25.

BREAK	LET	REREAD
CALL	LINE INPUT	RESET
CAUSE	MARGIN	RESTORE
CHAIN	MAT	RETRY
CLOSE	ON CONDITION	RETURN
CONTINUE	OPEN	REWRITE
DEBUG	PAUSE	SCRATCH
DELETE	PRINT	STOP
GET	PUT	SUBEXIT
GOSUB	RANDOMIZE	TRACE
GOTO	READ	WRITE
INPUT		

Figure 25. Imperative Statements

Example

```
:<+++##  >%%%
```

is equivalent, for numeric conversion, to

```
:++++##  %%%%
```

- The numeric value is converted according to the type of its conversion specification as follows:

  I-format

  The value is converted to an integer, rounding any fraction.

  F-format

  The value is converted to a fixed-point number, rounding the value or extending it with zeros in accordance with the conversion specifications.

  E-format

  The value is converted to a floating-point number, rounding the value or extending it with zeros in accordance with the conversion specification. The three or more ¬ characters (¬¬¬) in an E-format specification are used to indicate the print positions of the exponent part of a floating point number.

  The first ¬ character is replaced by the 'E'; the second by the sign of the exponent, '+' or '-'. The remaining ¬ characters are replaced by the value of of the exponent, which is right-justified with leading zeros.

IMAGE Specification	Result	Valid Exponent
¬¬¬	E±x	1 digit
¬¬¬¬	E±xx	2 digits
¬¬¬¬¬	E±xxx	3 digits

(where x is any digit)

Figure 26. IMAGE Statement Format Specification

If the exponent exceeds the width provided, an exception occurs.

- The converted numeric value is edited with respect to digit specifiers and commas as follows:
  - When % is the digit specifier, nonsignificant zeros are generated in the integer (or to the left of the decimal point).
  - When * is the digit specifier, nonsignificant zeros are replaced by asterisks.
  - When # is the digit specifier, nonsignificant zeros are suppressed.
  - When a comma appears in the image between groups of three digit specifiers, it will appear in the output where significant, provided at least one digit has been generated to the left of the position where the comma is to appear; if no digit has been generated to the left of the point of insertion, the comma is replaced by an

asterisk if the digit specifier is the *, or suppressed if the digit specifier is the #.

- If the number of digit specifiers is not adequate to contain all the significant digits, positions of the floating-header, if present, are used.

- A floating-header consists of a string of n repetitions of a +, -, or $ character, where n is greater than or equal to 1. The term "floating" is used because the "floating symbol," listed in Figure 27, "floats" from left to right among the n positions and appears in the output in a position dependent on the value and format of the numeric item, as explained further on.

Floating-Header	Sign of Value	Floating Symbol
+[+]...	positive	+
+[+]...	negative	-
-[-]...	positive	space
-[-]...	negative	-
$[$]...	positive	$
$[$]...	negative	$-

Figure 27. IMAGE Statement—Floating Symbol Usage

- Whenever a floating-header is not specified:

  - If the numeric value is negative, and if the conversion specification is large enough to contain the number and a minus sign, then the minus sign is placed immediately preceding the data.

  - If the number and the minus sign will not fit, then the entire specification is filled with asterisks.

  - If the value is positive, the value is displayed without a sign.

  - If the positive value does not fit, then the entire specification is filled with asterisks.

  Example

Value	Specification	Result
123	%%%%	0123
-123.45	****.##	-123.45
-1234	####	****
-12	####	b-12
-12	****	-*12
-12	%%%%	-012

  (The "b" shows a blank (space) position in the result.)

- If a floating-header is specified, the floating symbol is placed in the rightmost portion of the floating-header and to the left of the first digit. Any remaining leading positions of the floating-header are replaced with blanks.

  If there is insufficient room in the specification for the numeric value and for a nonblank floating symbol, then the entire specification is filled with asterisks.

When the floating header is a $, the numeric value is negative, and there is room (without termination of significant data) for both a $ and minus sign, a minus sign is inserted to the right of the $. If significant data would be truncated, the numeric specification is fill with asterisks.

Where the minus sign is displayed, when the floating header is $, depends upon whether or not zero suppression is in effect. (The # digit specifier does zero suppression):

- If # is the digit specifier, the minus sign floats and is inserted immediately to the left of the first significant digit.
- If % or * is the digit specifier, minus sign insertion depends on the number of $ positions specified, as follows:
  - If 2 or more $ are specified, the minus sign is inserted in a floating dollar position, immediately to the right of the $.
  - If at least 2 $ are not specified, the minus sign replaces the first digit specifier (* or %).

Example

Value	Specification	Result
123	---%%	123
123	+++%%	+123
12345	---%%	*****
12345	+++%%	*****
-123	$$$$####	bbb$-123
-12	$$$####	bb$b-12
-12	$$$****	b$-**12
-12	$$$%%%%	b$-0012
-12	$****	$-*12
-12	$%%%%	$-012
-12	$####	$b-12

(The "b" shows a blank (space) position in the result.)

Format Conversion Examples

The following are examples of the different types of conversion specifications. The letter "b" shows blank (space) in the result.

I-format Example

Value	Specification	Result
1000000	##,###,###	b1,000,000
-99999	##,###,###	bbb-99,999
3	****	***3
3.2	****	***3
100	***,***	****100
4	%%%%	0004

F-format Examples

Value	Specification	Result
12.145	###.##	b12.15
1000	***,***.##	**1,000.00
-77	%%%%.###	-077.000

E-format Examples

Value	Specification	Result
5	###.##¬¬¬¬	500.00E-02
-255.555	##.###¬¬¬	-2.556E+2

INPUT STATEMENT

The INPUT statement provides input through the terminal. (See also the INPUT FIELDS and INPUT FILE statements.)

Format

```
[MAT] INPUT [input-prompt :] input-list
            [err[,err]...]
```

Where:

input-prompt
can be one of the following:

PROMPT string expression

PROMPT quoted character string

quoted character string

input-list
is a list of variable or array names (possibly subscripted), to be input, separated by commas.

err
is one of the following:

EXIT line-ref

IOERR line-ref

CONV line-ref

SOFLOW line-ref

line-ref
is a line number, or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and the other err clauses are mutually exclusive.

Description

In interactive mode, this form of the INPUT statement allows the user to provide values for program variables from a terminal during program execution.

In batch mode, this form of the INPUT statement is system dependent. See the IBM BASIC Application Programming: System Services manual.

MAT KEYWORD: The MAT keyword preceding the INPUT keyword specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input-list.

See "Input/Output Lists" on page 70 for more information.

INPUT-PROMPT: If an input-prompt is present, the string expression will be displayed at the terminal during program execution as a prompt for the data to be entered.

An INPUT statement without an input prompt causes a question mark (?) to be generated on the terminal to indicate data is expected.

If the last previously executed PRINT statement had an output list, and the last item was followed by a semicolon (more data expected), the INPUT statement will cause any prompt, or question mark, to be appended to the data from the last PRINT statement and be sent to the terminal.

The terminal keyboard is then activated for input, without a return to the beginning of a new line. The user's expected response is to enter a list data values which will be assigned to the items in the input list. Each value must be of the same type, numeric or character, as the corresponding input list-item; however, a numeric value may be assigned to either a numeric or character variable.

INPUT-LIST: For input-lists consisting only of scalar variables, no assignment of values from the input reply takes place until an input reply line has been entered and checked for:

- The correspondence of each data item entered with the type of data item expected.
- The allowable range of values for each item to be within the limits.
- The number of items entered as exactly the number of items expected.

When an error is detected and the INPUT statement contains the corresponding error clause, control is transferred to the statement specified in the error clause. If there is no corresponding error clause, a request is made that the current input reply be re-entered.

For input lists that contain arrays (MAT), successive values are assigned as received from the input reply. If an error is detected as described in the scalar case, and the INPUT statement contains an appropriate error clause, control is transferred to the statement specified in the error clause. If there is no appropriate error clause, a request is made that the current input reply be reentered.

A significant difference between the scalar and MAT case is that in the scalar case no assignment is made for the entire input reply until all supplied values have been verified. In the MAT case, assignment is made for each item at the time that item is verified.

INPUT REPLY: Successive values entered at the terminal must be separated by commas. Consecutive commas cause the corresponding item of the input list to be passed over and to be left unchanged.

If the last character of the input reply is a comma, additional input is expected on the next line. The reply can be a "/", a value, or any other allowable response for the input list.

A "/" character at the end of a line of data causes any remaining items of the input list to be passed over and to be left unchanged.

If the current item of the input list is a scalar, one value is to be accepted for that item.

If the current item is an array, then a number of values corresponding to the number of elements in the array is accepted, and is assigned to members of the array with the rightmost array subscripts varying most rapidly.

Arrays in the input list may be redimensioned; the redimensioning occurs before values are assigned to the array.

In the input reply, the notation:

```
j*value
```

where j is a nonzero, unsigned integer constant, indicates that the value is to be assigned to the next j items of the input list; that is, j acts as a replication factor.

Example

```
5*555
```

specifies that the next 5 items of the input-list are to contain the value 555.

Character constants in the input reply are normally set off by quotation marks. However, these quotation marks may be omitted for character constants which:

- Contain no commas
- Have no leading or trailing blanks
- Contain no leading or trailing quotes
- Do not start with an integer immediately followed by an asterisk

All numeric data are rounded to a fixed number of significant digits, (or filled with zeros), 10 for integer data, and 17 for decimal data.

ERROR CONDITIONS: If an error clause is specified, the action taken is determined by the error option specified.

If an error clause is not specified, and if a numeric entry causes an overflow condition, a warning message is displayed and the runtime support requests that the line be reentered. If an underflow condition occurs, a warning message is displayed and the value is replaced by zero.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

All character data will have a length assigned which is equal to the length of the string transmitted; a string overflow occurs if the length of the string received exceeds the defined maximum length of the character string variable.

If a string overflow occurs, the SOFLOW exception occurs. If a numeric value cannot be converted as required, the CONV exception occurs. These errors can be handled by specifying the condition as an err clause, or by specifying an EXIT condition. The IOERR clause can also be specified to handle hardware malfunctions.

Example

```
100 INPUT "ENTER NAME:":NAME$
200 INPUT "ENTER HOME & BUSINESS PHONE:":HOME$,BUS$
```

The above statements will prompt for name, then home and business phone numbers.

INPUT FIELDS STATEMENT (FOR FULL SCREEN TERMINAL INPUT)

The INPUT FIELDS statement reads one or more data values from one or more specific fields of the terminal screen and assigns the value(s) to one or more variables.

Format

```
INPUT [#fileref[,]] FIELDS field-definition:
          input-list [err[,err]...]
```

Where:

fileref
is a numeric expression whose rounded integer value evaluates to zero.

field-definition
can be:

character expression

or

MAT character array name

Each character expression or character array name must evaluate to:

"row, column,data-form[,[leading][,trailing]]"

Where:

row
is a positive nonzero integer, specifying the screen row of the field.

column
is a positive nonzero integer, specifying the column of the first character in the field.

data-form
can be one of the data forms shown in Figure 28 on page 162.

leading
are display and control attributes for the input field

trailing
are display attributes for the positions between the input field and the next field and are control attributes for this input field.

Display attributes that have meaning to IBM BASIC are:

H
highlighted

I
invisible (not displayed)

N
normal intensity

Note: For ease of migration from other BASIC products, B, R, and U are also accepted and treated as N (normal intensity). Multiple attributes can be specified. If I

Data Form	Meaning
W	Length of data item.
C[W]	Character data.
V[W]	Character data with trailing blanks removed on input.
NW[.d]	Conversion of numeric data from character data.
G[W[.d]]	Represents either character data or conversion of numeric data from character data depending upon whether the type of the receiving field is character or numeric.

Where:

W
is an unsigned, nonzero integer constant, which may optionally be preceded with blanks.

d
is an unsigned integer constant, which must be less than or equal to w.

Note: The total length of w, in characters, can be from 1 through (screen-size - 2) for character data, or from 1 through 156 for numeric data. (The screen size is the total number of characters on the screen.) If w (or w.d) is omitted, the length is 1 character.

Figure 28. INPUT FIELDS Statement—Data Form Codes

is specified, it overrides both H and N; H overrides N. N is the default.

Leading control attributes that have meaning to IBM BASIC are:

A
automatic field exit (when a character is entered into the last position of the field, the cursor automatically advances to the first character position in the next input field). If not specified, the cursor advances to the next non-attribute character after the field.

C
position the cursor to this field first

The trailing control attribute that has meaning to IBM BASIC is:

A
automatic field exit (when a character is entered into the last position of the field, the cursor automatically advances to the first character position in the next input field). If not specified, the cursor advances to the next non-attribute character after the field.

input-list
is a list of one or more variables, array elements, and/or entire arrays (prefaced with MAT). List elements are separated by commas.

err
can be one of the following:

EXIT line-ref

CONV line-ref

IOERR line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

When the INPUT FIELDS statement is executed, the terminal user can enter a value for each field specified in the statement. At the beginning of execution, the cursor is positioned to one of the following:

- The first character of the first (or only) field specified
- If the C attribute is specified, to the first character of the last (or only) field with C as the leading attribute.

and BASIC waits for value(s) to be entered.

After entering the last character of a field, subsequent cursor positioning depends upon whether or not control-attribute A is specified:

- If A is specified, the cursor is positioned at the beginning of the next field on the terminal screen.
- If A is not specified, the cursor is positioned to the first non-attribute character after the field.

The terminal user can position the cursor by using the cursor positioning keys (including the NEXT FIELD and PREVIOUS FIELD keys).

All field-definitions are syntax-checked before the screen is altered or any data transfer takes place.

No data transfer takes place until the terminal user presses ENTER. (Pressing a PF key does not enter the data, which remains on the screen. If SKEY is set to SYSTEM or IGNORE, pressing the PF key has no effect at all. If SKEY is set to GOTO, control is transferred to the line specified.)

When the data is transferred, multiple input fields are processed in the same order that the fields are defined in the field-definition array. As each field is processed, the data value is assigned to the corresponding input-list item. Unlike the INPUT statement, in which all data values are verified before any data is transferred, the INPUT FIELDS statement verifies each data value as it is transferred. The order that the fields are assigned in the field-definition array corresponds to the order in which the input-list items are defined. (That is, the first field-definition corresponds to the first input-list item, the second field-definition corresponds to the second input-list item, and so on.)

At the completion of execution, the number of input-list items successfully transferred can be obtained through the CNT intrinsic function.

If the terminal does not have a screen, an IOERR exception occurs. (See "Full Screen Input/Output Statements" on page 73.)

FILEREF: The fileref is a numeric expression that should, when rounded to an integer, evaluate to zero; if it does not, an IOERR exception occurs. The standard system action is to replace the value with zero.

FIELD-DEFINITION: A field-definition entry can be a character expression or MAT character array name:

- If a field-definition entry is a character expression, it defines one input field, and only one item of data can be entered.
- If a field-definition entry is MAT character array name, it can define one or more input fields. In this case, the field-definition entry must be a one dimensional array; the field-definition entries within the array need not match the order of the fields on the screen.

If an array is specified for a field-definition entry, the number of fields is the number of input-list items, not the number of elements in the array. The number of elements in the array can exceed the number of input-list items; any extra array elements are ignored. However, all the array elements are syntax checked.

Row and column are positive, nonzero integers that specify the starting location of each field. Row 1, column 1 is the upper left-hand corner of the screen. An exception occurs if either row or column exceeds the dimensions of the screen.

Input fields cannot overlap; they may not contain attribute fields created by a previous PRINT FIELDS statement.

Data-Form specifies the length and data type of the data to be entered and any data conversions to be performed. Figure 28 on page 162 shows the data forms allowed.

The data-form specifies the number of characters in the field. Fields that extend beyond the rightmost column are continued in column one of the next row, the bottom row continuing to the top of the screen with wraparound.

For the C, V, and G data forms, the length of the field (w or w.d) may be omitted from the field-definition by omitting the length in the data-form specification. If the length is omitted, the field is one character long.

Display Attributes specify how the display is treated.

Leading Display Attributes specify how the input field is to display on the screen. The leading attribute occupies a character position on the screen preceding the field. The leading attribute unprotects the field.

Trailing Display Attributes specify how the positions between the input field and the next field are to display. The trailing attribute occupies a character position on the screen following the field. The trailing attribute protects the following field.

The location of the trailing display attribute for one field can overlap with the leading display attribute of the following field. If leading and trailing attributes overlap, the last attribute written to the screen is the one in effect.

Control Attributes specify actions to be taken for each field.

Leading Control Attributes specify how the input field is to be treated:

A An automatic field exit occurs if the terminal user places a character in the last position of the field.

C Places the cursor at the beginning of the specified field. If C is specified for more than one field in an array, the cursor is placed at the beginning of the last field specified with the C attribute.

The Trailing Control Attribute can be specified as A (for automatic field exit); it applies to the preceding field.

Any combination of leading display and control attributes is allowed, and any combination of trailing display attributes is allowed. If any character other than those given above is specified, it is ignored.

A set of leading or trailing attributes should not be separated by commas; the comma specifies the beginning and ending of each leading or trailing list. The attributes can be entered in any order.

INPUT-LIST: There must be at least one entry in the input-list.

For each variable name or array element in the input-list, only one item of data can be entered.

If the input-list is MAT array name, data values are placed into the array on a row-by-row basis when the values are transferred from the screen.

ERROR CONDITIONS: If a string overflow occurs, the SOFLOW exception occurs. If a numeric conversion cannot be performed as required, the CONV exception occurs. If a hardware malfunction prevents completion of the input process, the IOERR exception occurs.

These exceptions can be recovered from, if the CONV, IOERR, or SOFLOW clauses are specified, or if an EXIT clause refers to an EXIT statement that contains these clauses.

The I/O error conditions interact with the ON Condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example 1

```
110 INPUT FIELDS "10,12,C15,I": PASSWORD$
```

Starting in row 10, column 12, 15 characters are read from a field into the variable PASSWORD$. The entered characters are not displayed.

Example 2

```
100 A$="22,5,N2"
110 INPUT FIELDS A$ : AGE%
```

Reads a 2-character numeric constant starting in row 22, column 5, into the variable AGE%. Intensity is not specified, so the default NORMAL is in effect.

Example 3

```
100 OPTION BASE 1
110 DIM A$(4), B$(4), NAME$*30, ADDR$*30, CITY$*30,&
  & ST_ADDR_CODE$*30
130 DATA "10,10,C10,H", "12,20,C10",&
  & "14,20,C10", "16,20,C20,N"
140 MAT READ A$
150 DATA "10,45,C30,HA,H","12,45,C30,HA,H",&
  & "14,45,C30,HA,H","16,45,C30,H,N"
160 MAT READ B$
170 PRINT NEWPAGE
180 PRINT FIELDS MAT A$: "NAME", "ADDRESS", "CITY",&
  & "STATE, ADDRESS-CODE"
190 INPUT FIELDS MAT B$: NAME$, ADDR$, CITY$, ST_ADDR_CODE$
200 PRINT FIELDS "20,15,C15,H": "OK? TYPE Y OR N"
210 INPUT FIELDS "20,31,C1,H,N": RETRY$
220 IF UPRC$(RETRY$) = UPRC$("N") THEN 190
230 PRINT NEWPAGE: PRINT "YOU ENTERED"
240 PRINT NAME$: PRINT ADDR$
250 PRINT CITY$: PRINT ST_ADDR_CODE$
260 END
```

This series of statements sets up two arrays of field-definitions, A$ and B$, each with 4 elements.

When the first PRINT FIELDS statement (180) is executed, the headings are displayed on the screen in rows 10, 12, 14, and 16, and all at column 20.

When the first INPUT FIELDS statement (190) is executed, the cursor is positioned first at row 10, column 45. When the first field is filled in by the terminal user, the cursor advances to the beginning of the next field (row 12, column 45). When this field is filled in, the cursor is positioned at row 14, column 45, and, after this field is filled, at row 16, column 45.

When the terminal user presses ENTER, the input data on the screen is transferred to NAME$, ADDR$, CITY$, and ST_ADDR_CODE$ in that order. Note that not all fields need be filled before ENTER is pressed.

Statements 200 through 220 serve as a check that the user hasn't pressed ENTER by mistake. If the user wishes to reenter the fields, then the program loops back to statement 190 for another try.

When the user indicates that the entry has been correctly made, a new screen displays the data values entered.

INPUT FILE STATEMENT

The INPUT File statement reads either display or internal files.

```
Format 1 (display files)

    [MAT] INPUT #fileref  [[,] input-prompt]
                :input-list[,SKIP REST]
                [err[,err]...]

Format 2 (internal files)

    [MAT] INPUT #fileref :input-list[[,], SKIP REST]
                [err[,err]...]
```

Where:

fileref
is a numeric expression which when evaluated and rounded, must be a positive integer within the range 0 to 255, and which identifies the file to be read.

input-prompt
is a quoted character string.

Note: The fileref and input-prompt may occur in either order.

input-list
is an input list of variable or array names (possibly subscripted) separated by commas.

err
is one of the following:

EXIT line-ref

EOF line-ref

IOERR line-ref

CONV line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

The INPUT file statement, when referring to a fileref connected to a display format file, receives data in a manner similar to the way the INPUT statement receives data from a terminal; a record from the file is accessed, and its contents are processed as if they represented a line entered on the terminal in response to an INPUT request. However, there are exceptions to this similarity.

- During input from a file, the input-prompt has no meaning and is ignored.
- Unlike input from a terminal, which can request a new reply if the current reply is invalid, file input, when it retrieves an

invalid record, causes an exception. An exception can occur in the following situations:

1. An attempt is made to read a character data item into a numeric variable.
2. The number of data values does not match the number of variables in the input list.
3. A string or numeric overflow occurs.

When the file reference is 0, this statement acts like the INPUT statement for the terminal. (See "INPUT Statement" on page 158.)

When operating on a stream file, the INPUT file statement acts like a GET statement. (See "GET Statement" on page 143.)

When operating on a record-oriented internal-format file, the INPUT File statement acts like a READ statement. (See "READ FILE Statement" on page 237.)

MAT KEYWORD: The MAT keyword preceding the INPUT keyword specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input list.

See "Input/Output Lists" on page 70 for more information.

FILEREF: The fileref must refer to a display or internal file. (See "Combinations of File Organization and Format" on page 57.)

INPUT-LIST: If the input list does not specify enough variables to accommodate all of the values retrieved from a record of a display or internal format file, a CONV error occurs. To keep the CONV condition from occurring in this case, the SKIP REST clause can be specified. The SKIP REST clause causes excess data to be ignored, thus not causing the CONV condition.

See "Input/Output Lists" on page 70 for additional considerations.

ERROR CONDITIONS: If a numeric entry causes an overflow condition, a warning message is displayed and a request is made that the line be entered.

If an underflow condition occurs, a warning message is displayed and the value is replaced by zero.

All character data has a length assigned which is equal to the length of the string transmitted; a string overflow occurs if the length of the string received exceeds the defined maximum length of the variable.

If a string overflow occurs, the SOFLOW exception occurs. If a numeric value cannot be converted as required, the CONV exception occurs.

In addition, an EOF clause may be specified to process the condition which occurs when an attempt is made to access another record when end-of-file has been reached.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 INPUT #5 : A$,B% SOFLOW 500,CONV 600
```

In the above example, assuming a display file is associated to file reference 5, a record is read and values assigned to A$ and B% as if an INPUT statement were executed. If a string overflow error occurs in assigning data to A$, control passes to line number 500; if a numeric conversion error occurs with B%, control passes to line number 600.

invalid record, causes an exception. An exception can occur in the following situations:

1. An attempt is made to read a character data item into a numeric variable.
2. The number of data values does not match the number of variables in the input list.
3. A string or numeric overflow occurs.

When the file reference is 0, this statement acts like the INPUT statement for the terminal. (See "INPUT Statement" on page 158.)

When operating on a stream file, the INPUT file statement acts like a GET statement. (See "GET Statement" on page 143.)

When operating on a record-oriented internal-format file, the INPUT File statement acts like a READ statement. (See "READ FILE Statement" on page 237.)

MAT KEYWORD: The MAT keyword preceding the INPUT keyword specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input list.

See "Input/Output Lists" on page 70 for more information.

FILEREF: The fileref must refer to a display or internal file. (See "Combinations of File Organization and Format" on page 57.)

INPUT-LIST: If the input list does not specify enough variables to accommodate all of the values retrieved from a record of a display or internal format file, a CONV error occurs. To keep the CONV condition from occurring in this case, the SKIP REST clause can be specified. The SKIP REST clause causes excess data to be ignored, thus not causing the CONV condition.

See "Input/Output Lists" on page 70 for additional considerations.

ERROR CONDITIONS: If a numeric entry causes an overflow condition, a warning message is displayed and a request is made that the line be entered.

If an underflow condition occurs, a warning message is displayed and the value is replaced by zero.

All character data has a length assigned which is equal to the length of the string transmitted; a string overflow occurs if the length of the string received exceeds the defined maximum length of the variable.

If a string overflow occurs, the SOFLOW exception occurs. If a numeric value cannot be converted as required, the CONV exception occurs.

In addition, an EOF clause may be specified to process the condition which occurs when an attempt is made to access another record when end-of-file has been reached.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 INPUT #5 : A$,B% SOFLOW 500,CONV 600
```

In the above example, assuming a display file is associated to file reference 5, a record is read and values assigned to A$ and B% as if an INPUT statement were executed. If a string overflow error occurs in assigning data to A$, control passes to line number 500; if a numeric conversion error occurs with B%, control passes to line number 600.

INTEGER STATEMENT

The INTEGER statement specifies which identifiers are to be assigned integer type.

Format

```
INTEGER [[identifier|(letter-list)]...]
```

Where:

identifier
may be a specific numeric identifier.

letter-list
is a list of letters and/or ranges of letters separated by commas. A range of letters is represented by the first and last letters in the range separated by a minus sign.

For compatibility with other BASICs, the keyword DEFINT (define integer) may be used in place of INTEGER. The syntax and semantics of the DEFINT statement are the same as those for the INTEGER statement.

Description

The INTEGER statement declares a specific identifier, or any identifier beginning with a specific letter, as having integer type; or, when used without a list, it specifies the default type for all identifiers not otherwise typed in a program unit.

INTEGER statements may appear anywhere in a program unit, and affect identifiers throughout the program unit. The identifiers affected are variable names, array names, or function names.

TYPE SPEC: If the INTEGER statement specifies a parenthetical list of letters, all identifiers beginning with these letters are to be typed integer, unless they end in a contradictory self-typing character "#" or "$", or unless they are explicitly declared in a DECIMAL statement by identifier.

The letter-list may be specified as either single letters (A, B, C, D) or as a series of consecutive letters, such as (A-D, X-Z), indicating A through D and X through Z.

An identifier explicitly stated in an INTEGER statement may end with the self-typing character "%" but not with "#" or "$".

If an INTEGER statement specifies no identifiers and no letter-list, the default type for all identifiers in the program-unit is set to integer.

If an INTEGER (or DECIMAL) typing statement is not specified, the default typing is DECIMAL.

Example 1

```
100 INTEGER ABLE,(C-E,G,J,L),NANCY
```

specifies that identifiers ABLE and NANCY, as well as all identifiers beginning with the letters C, D, E, G, J, and L are typed integer. If the program unit subsequently contains a variable named GEORGE, it would be assigned integer type; however, CHARLIE# would be assigned decimal and EDGAR$ assigned character.

Example 2

```
100 DEFINT ABLE, (C-E,G,J,L),NANCY
```

is equivalent to Example 1.

Immediate Execution

Integer type is valid for immediate variables and arrays. The immediate INTEGER statement has the same form as when used in a program.

See "Immediate Statements" on page 260 and "Immediate Type and Dimensions" on page 262 for a discussion of the interaction of immediate INTEGER statements with other immediate statements and program statements.

LET (SCALAR ASSIGNMENT) STATEMENT

The LET scalar assignment statement assigns values to both numeric and character variables.

Format

```
[LET] variable [, variable]...=expression
```

Where:

variable
is:

a numeric variable.

a character variable (with optional substring notation).

a subscripted numeric array member.

a subscripted character array member (with optional substring notation).

expression
is any numeric expression (for numeric variables) or character expression (for character variables).

Description

The value of the numeric or character expression to the right of the equals sign is evaluated and assigned to the numeric or character variable on the left of the equals sign.

The keyword LET is optional. In the following example, the statements are equivalent.

Example

```
100 LET SUMM = ADD1+ADD2+ADD3
110 SUMM = ADD1+ADD2+ADD3
```

VARIABLE: The type of the variable(s) on the left side of the equals sign must agree with the type of the expression on the right side. They both must be either character or numeric. However, in the case of numeric LET statements, the left side can be integer when the right side is decimal, and vice versa.

A single value may be assigned to several variables at once using multiple variables to the left of the equal sign.

Example

```
100 LET A$,B$,C$ = 'TOTAL'
```

is functionally equivalent to:

```
100 LET A$ = 'TOTAL'
110 LET B$ = 'TOTAL'
120 LET C$ = 'TOTAL'
```

Multiple assignments are made from left to right.

```
100 LET I, A(I) = 5
```

is functionally equivalent to:

```
100 LET I = 5
110 LET A(I) = 5
```

EXPRESSION: For numeric assignment statements where the type (integer or decimal) of the expression to the right of the equal sign does not agree with that of the variable to the left of the equals sign, the result of the expression is converted to the type of the variable. Rounding occurs when converting decimal to integer.

Example

```
100 LET SUMM% = ADD# + ADC#
```

If the value of ADD# is 20.7 and ADC# is 10.0, the value of SUMM% after the execution of the LET statement is 31.

Immediate Execution

The LET immediate statement assigns the expression on the right of the equal sign to the variable(s) on the left.

Immediate and program variables may be used in expressions. There are two restrictions:

1. Immediate LET statements cannot refer to user-defined functions defined in a program (DEF statements). However, intrinsic functions may be used.
2. The keyword LET is optional unless the immediate LET statement assigns a value to a variable having the same name as one of the IBM BASIC commands. See "IBM BASIC Commands" on page 267.

Example

```
100 LIST = 3          is an error

100 LET LIST = 3      is accepted
```

See "Immediate Statements" on page 260 for additional information on immediate execution.

LINE INPUT/LINPUT STATEMENT

The LINE INPUT statement allows the unformatted input of character strings from a terminal.

Format

```
[MAT] LINE INPUT [input-prompt:]
      input-list [err[,err]]
```

Where:

input-prompt
can be:

PROMPT string expression

or

a quoted character string

input-list
is an input list of character items.

err
is one of the following:

EXIT line-ref

IOERR line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and the other two err clauses are mutually exclusive.

Description

LINE INPUT may also be spelled LINPUT.

Character strings which contain commas and other characters usually considered as delimiters can be entered from a terminal with the LINE INPUT statement.

MAT KEYWORD: The MAT keyword preceding the LINE INPUT keywords specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input list.

See "Input/Output Lists" on page 70 for more information.

INPUT-PROMPT: If the statement specifies an input-prompt, the result of the PROMPT string expression or the quoted character string is displayed.

If a prompt is not present, a question mark is displayed.

If the last PRINT statement had an output list in which the last data item was followed by a semicolon (more data expected), the input statement causes any prompt, or the question mark, to be appended to the data from the last PRINT statement and be sent to the terminal.

Example

```
100 PRINT "PROM";
110 LINE INPUT 'PT': A$
     .
     .
     .
```

When these statements are executed, the display terminal displays the following:

```
PROMPT _
```

(The underscore indicates where the cursor is positioned for the terminal user to enter a line of input.)

The terminal is then activated for input, without a return to the beginning of a new line. The response is to enter one line for each variable or array element in the input list. A question mark prompt is issued for each subsequent line.

The entire contents of successive input lines are assigned to the string variables in the input list. Array elements are assigned with the rightmost subscripts varying most rapidly. A new question mark prompt is issued for each variable or array element after the first.

If redimensioning is specified, then dynamic redimensioning takes place before values are assigned to the redimensioned array.

ERROR CONDITIONS: The error conditions IOERR (input/output error) and SOFLOW (string overflow) may be recoverable if an err clause for the condition is specified in the statement or on the referenced EXIT statement.

SOFLOW occurs if the string entered at the terminal is longer than the maximum length of the corresponding character variable.

IOERR occurs if a hardware malfunction prevents the completion of the input process.

The I/O error conditions interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 LINE INPUT "TYPE IT" : A$,B$
```

The prompt message, TYPE IT, appears on the terminal with the cursor positioned immediately to the right.

If AB,C110?124 is entered, that data is assigned to A$ having a length of 11. The terminal then prompts (?) for the second value to be stored in B$.

LINE INPUT/LINPUT FILE STATEMENT

The LINE INPUT File statement allows the unformatted input of data from a file.

Format

```
[MAT] LINE INPUT #fileref
      [[,]input-prompt]:input-list
      [err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 0 to 255, and which identifies the file to be processed.

input-prompt
is a quoted character string.

Note: The fileref and input-prompt clauses may occur in any order.

input-list
is an input list of character variables or array names (possibly subscripted).

err
is one of the following:

EXIT line-ref

EOF line-ref

IOERR line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

LINE INPUT may also be spelled LINPUT.

The LINE INPUT File statement processes records of a file the same way the LINE INPUT statement processes lines of data from a terminal. The entire contents of each successive record, including commas and other characters usually thought of as delimiters, is assigned to each successive character string variable in the input list.

MAT KEYWORD: The MAT keyword preceding the LINE INPUT keywords specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input list.

See "Input/Output Lists" on page 70 for more information.

Array elements are assigned with the rightmost subscripts varying most rapidly.

If redimensioning is specified, dynamic redimensioning takes place before values are assigned to the redimensioned array.

FILEREF: A file which is accessed with this statement must have display format. The length of the records must not exceed the length of the corresponding character variables in the input list. (See "Combinations of File Organization and Format" on page 57.)

INPUT-PROMPT: The input-prompt clause is ignored if the file reference number is not zero. When fileref is zero, the terminal is accessed and this statement acts identically to a LINE INPUT statement (see "LINE INPUT/LINPUT Statement" on page 173).

ERROR CONDITIONS: The EXIT, IOERR, and SOFLOW err clauses function as they do for a terminal. In addition, the EOF err clause can be used to process the condition caused by attempting to access another record when the end of the file has been reached.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 LINPUT #3 : A$
```

Retrieves a record from the file reference number 3 and assigns the record to A$.

If redimensioning is specified, dynamic redimensioning takes place before values are assigned to the redimensioned array.

FILEREF: A file which is accessed with this statement must have display format. The length of the records must not exceed the length of the corresponding character variables in the input list. (See "Combinations of File Organization and Format" on page 57.)

INPUT-PROMPT: The input-prompt clause is ignored if the file reference number is not zero. When fileref is zero, the terminal is accessed and this statement acts identically to a LINE INPUT statement (see "LINE INPUT/LINPUT Statement" on page 173).

ERROR CONDITIONS: The EXIT, IOERR, and SOFLOW err clauses function as they do for a terminal. In addition, the EOF err clause can be used to process the condition caused by attempting to access another record when the end of the file has been reached.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 LINPUT #3 : A$
```

Retrieves a record from the file reference number 3 and assigns the record to A$.

LOOP STATEMENT

The LOOP statement delimits a DO loop.

Format

```
LOOP  [{WHILE|UNTIL} logical expression]
```

Where:

logical expression
can be any logical expression as documented in in "Logical Expressions" on page 31.

Description

The LOOP statement specifies the end of a DO loop.

The WHILE/UNTIL clause is the exit condition. When the exit condition is satisfied, processing continues with the next executable statement.

The exit condition is satisfied if:

- The value of the logical expression following WHILE is false
- The value of the logical expression following UNTIL is true

The LOOP statement must always follow a matching DO statement.

See "DO Statement" on page 116 and "Loop Control Statements" on page 62.

MARGIN STATEMENT

The MARGIN statement specifies where output may begin and end on a terminal screen or page.

Format

```
MARGIN numeric expression

   or

MARGIN [RIGHT numeric expression]
       [LEFT numeric expression]
       [TOP numeric expression]
       [BOTTOM numeric expression]
```

Where:

numeric expression
can be any numeric expression

BOTTOM
defines the bottom margin of the page

LEFT
defines the left margin of the page

RIGHT
defines the right margin of the page

TOP
defines the top margin of the page

Note: The keywords BOTTOM, LEFT, RIGHT, and TOP may appear in any order.

Description

The MARGIN statement is an executable statement which sets the boundaries for subsequent PRINT statement output to a terminal.

The first format of the statement may be used to set the right margin; in the second format, at least one clause must be specified.

NUMERIC EXPRESSION: The rounded integer values of the numeric expressions are used to determine:

- Where the last line of the screen or page may appear (BOTTOM)
- Where the first position of a line may appear (LEFT)
- Where the last position of a line may occur (RIGHT)
- Where the first line of the screen or page may appear (TOP)

If any of the following margin rules are violated, an exception occurs.

MARGIN LEFT: The value specified for the left margin must be within the range 1 to the defined line width of the terminal, and must not exceed the value of the right margin.

If no left margin is specified, the default value of 1 is assigned.

MARGIN RIGHT: The value specified for the right margin must be within the range 0 to the defined line width of the terminal. If a value of zero is specified, or if no right margin is specified, the default value of the terminal's line width is assumed.

For PRINT statements without IMAGE or FORM control, the right margin, if not zero, must exceed the left margin by an amount necessary to print a numeric value from an unformatted print in floating decimal form. For SPREC that value is 13; for LPREC it is 19.

MARGIN TOP: The value specified for the top margin must be within the range 1 to the value of the bottom margin, unless the bottom margin is 0.

If no top margin is specified, the default value of 1 is assumed.

MARGIN BOTTOM: The value specified for the bottom margin must be within the range 0 to 32,767.

If no bottom margin is specified, the default value of 0 is assumed.

A bottom margin of 0 implies no line limit, and no ENDPAGE (or PAGEOFLOW) condition is to be generated.

GENERAL CONSIDERATIONS: The MARGIN statement may appear as often as necessary within a program to provide the desired formats. Execution of a MARGIN statement provides new parameters immediately.

Example

```
MARGIN LEFT 1 RIGHT 65 TOP 7 BOTTOM 60
```

are valid margins for an output file on a hardcopy terminal with 8-1/2 x 11-inch pages.

LEFT AND RIGHT MARGINS: The LEFT and RIGHT MARGIN parameters interact with the various forms of the PRINT statement, as follows:

PRINT	The print-zone size, which is established by default or the OPTION PRTZO statement, is counted from the LEFT margin. If the left margin is 10 and the print-zone size is 20, the print zones will be located at columns 10,30,50, etc. The RIGHT margin determines the location of the last print zone. There must be enough columns between the beginning of the last print zone and the right margin to satisfy the print-zone size. In the above example, if the right margin was 80, the last print zone would start at column 50.
PRINT USING IMAGE	The image begins at the LEFT margin. An image that extends beyond the RIGHT margin causes a CONV exception when the PRINT statement is processed.
PRINT USING FORM	If the FORM refers to a column position beyond the LEFT or RIGHT margins, a CONV exception occurs.

TOP AND BOTTOM MARGINS AND THE ENDPAGE CONDITION: For display terminals, PRINT lines scroll continuously onto the screen from the bottom. In order to insure that lines are not scrolled off the top of the screen before they are seen, BASIC does not allow more than the number of lines that fit on the screen to scroll without a positive response from the terminal user. For example, on a 3278

model 2 terminal which holds 24 lines on its screen, as soon as 23 print lines have scrolled onto the screen and the 24th is printed by the program, the scrolling stops and the bottom line (the 24th) displays the message

```
**ENTER TO CONTINUE**
```

This means the user must press the ENTER key to allow the scrolling, and the program, to continue. When ENTER is pressed, the first line scrolls off the top and the 24th line scrolls onto the bottom of the screen.

On a hard-copy terminal with printer characteristics, the TOP and BOTTOM margins can be used to control when page ejects (top-of-form) are generated and how many blank lines are automatically generated after a page eject. These page ejects correspond to ENDPAGE exceptions generated by PRINT statements. On a display terminal screen, the **ENTER TO CONTINUE** scrolling stops can be controlled by ENDPAGE exceptions.

When the number of lines printed since the last ENDPAGE exception (or since the start of the program or since the last NEWPAGE in a print list or PAGE in a FORM) is equal to the BOTTOM margin, and the action associated with ENDPAGE is SYSTEM (see "ON Condition Statement" on page 203), the ENDPAGE exception is generated immediately, even if the program is in the middle of a PRINT statement which prints more than one line. The SYSTEM action is to:

- On a display terminal: Print as many blank lines as necessary to cause scrolling to halt and the **ENTER TO CONTINUE** prompt to appear. Then, when ENTER is pressed, to print as many blank lines as specified by TOP minus one. Then continue execution of the program.
- On a hard-copy terminal: Eject a page, print TOP minus one blank lines, and continue execution of the program.

If the action associated with NEWPAGE is GOTO (the program takes control of the exception), the exception is not generated until the generating PRINT statement is completed. Thus if the PRINT statement produces more than one line, more than one BOTTOM lines are printed before the transfer of control occurs.

MARGIN FILE STATEMENT

The MARGIN File statement specifies the page margins for display-format files being accessed by PRINT File statements.

Format

```
MARGIN #fileref numeric expression

   or

MARGIN #fileref [RIGHT numeric expression]
       [LEFT numeric expression]
       [TOP numeric expression]
       [BOTTOM numeric expression]
```

Where:

fileref
is a numeric expression which when evaluated and rounded, must be a positive integer within the range of 0 to 255.

numeric expression
can be any numeric expression

BOTTOM
defines the bottom margin of the page

LEFT
defines the left margin of the page

RIGHT
defines the right margin of the page

TOP
defines the top margin of the page

Note: The keywords RIGHT, LEFT, TOP, and BOTTOM may appear in any order. At least one must appear.

Description

The MARGIN File statement functions for a display format file the way the MARGIN statement functions for a terminal. The margins dictate what the records of the file created by PRINT File statements would look like if directed to a display output device. The effects of this statement remain until another MARGIN File statement explicitly change one or more margins of that file, or until the file is closed.

FILEREF: The fileref must refer to a display file. (See "Combinations of File Organization and Format" on page 57.)

If the following margin rules are violated, an exception occurs.

MARGIN LEFT: The value specified for the left margin must be within the range 1 to the record length of the file, and must not exceed the value of the right margin (unless the right margin is 0).

The default value for the left margin is 1.

MARGIN RIGHT: The right margin must be within the range 0 to the record length of the file.

If no value is specified, the default is the lesser of 133 and the record length.

A right margin of 0 implies there is no limit, within the record length of the file.

For PRINT statements without FORM or IMAGE control, the right margin, if not zero, must exceed the left margin by an amount necessary to print a numeric value from an unformatted print in floating-point form. For SPREC that value is 13, for LPREC it is 19.

MARGIN TOP: The top margin must be within the range 0 to the bottom margin.

The default value of 1 is assumed if no value is specified.

Whenever an ENDPAGE condition occurs on the file, the top margin less 1 indicates how many blank records are to be generated before the next data record.

MARGIN BOTTOM: The bottom margin must be within the range 1 to maximum record size for the file.

The default value is an installation parameter. The IBM supplied default is 60.

A bottom margin of 0 implies no line limit, and no ENDPAGE condition is generated.

GENERAL CONSIDERATIONS: The MARGIN file parameters interact with the various forms of the PRINT File statement in the same manner as the MARGIN statement interacts with the PRINT statement. (See "MARGIN Statement" on page 178.)

If fileref is zero, the MARGIN File statement acts as a MARGIN statement and sets the margins for the terminal.

MAT (ARRAY ASSIGNMENT) STATEMENT

The MAT statement assigns values and dimensions to an array.

Format

```
MAT arrayname = array expression
                [(redimension)]
```

Where:

arrayname
is an array name.

array expression
is one of the following:

```
arrayname1
(expression)
arrayname1+arrayname2
arrayname1-arrayname2
arrayname1*arrayname2
(expression)*arrayname1
arrayname1&arrayname2
(expression)&arrayname1
arrayname1&(expression)
[(expression)*]IDN
ZER
[(expression)*]CON
NUL$
INV(arrayname1)
TRN(arrayname1)
AIDX(arrayname1)
DIDX(arrayname1)
ASORT(arrayname1)
DSORT(arrayname1)
```

arrayname1 and arrayname2
are arraynames (see description for restrictions on type and dimensions).

expression
is a numeric or string expression (see description for restrictions on type).

redimension
is one to seven numeric expressions separated by commas.

Description

Execution of an array assignment statement causes the array expression to be evaluated and its value assigned to the array named to the left of the equals sign.

ARRAY EXPRESSION: If the right side of the equal sign is an array expression, it is evaluated and an element-by-element assignment is made to the receiving array, on the left of the equal sign.

Example 1

```
MAT ARRAX = VALA+VALB
```

Each member of ARRAX is assigned the sum of the corresponding values of arrays VALA and VALB, that is, the value in VALA(0) is added to the value in VALB(0) and the sum is placed in ARRAX(0).

If the right side of the equals sign is a parenthesized scalar expression, it is evaluated and each element of the receiving array is set to that value.

Example 2

```
MAT AR% = (4*10.02+8/3)
```

Every element in the array AR% is set equal to 43.

When the array is a character array, the character assigned may not exceed the defined maximum length (either explicit or implicit definition) of the elements of the character array. If there is an attempt to assign character data that exceeds this length, a string overflow occurs.

All array statements which result in assignment (with the exception of AIDX and DIDX) may have expressions on the right side of the equal sign that have the same array name as the one on the left. This is termed "in-position" replacement. Both of the following statements are permitted.

Example 3

```
100 MAT A = A + A
110 MAT A = INV(A)
```

If an error occurs, such as numeric overflow or string overflow, an exception is generated at the point of the error. This means that the array assignment may be partially complete (part of the array on the left side may have been changed). In order to allow recovery from such situations, IBM BASIC assures that the operands of the right side expression have not been altered when the exception occurs. If the array on the left appears as an operand on the right (including as an array element in a scalar expression), the right side is evaluated into a temporary array and then moved to the left side array.

For numeric MAT statements where the type (integer or decimal) of the array expression does not agree with that of the array to the left of the equal sign, the result of the expression is converted to the type of the array on the left. If necessary, rounding occurs prior to the assignment.

REDIMENSION CLAUSE: When the redimension clause is specified, this array is redimensioned dynamically; that is, the number of dimensions and the size in each dimension is changed to match the number of dimensions and the size of each dimension specified by the values in the redimension specification on the right.

Example 4

```
120 DIM A(100), B(3,4), C(4,3)
130 MAT A = B*C !A redimensioned to (3,3)
140 MAT A = C*B !A redimensioned to (4,4)
150 MAT A = A(2,2) !A redimensioned to (2,2)
```

When an array is redimensioned dynamically, the upper bounds are changed to match the size of its new value and the current lower bounds as defined by the OPTION BASE are retained. The new bounds need not be individually less than or equal to the bounds specified in the dimension statement for that array (or by the default dimensions if the array is not dimensioned explicitly), as long as the total number of elements for the redimensioned array does not exceed the total number of elements specified by the array's original dimensions. What this means is that for array B in the above example, you can change any one of the dimensions or even add dimensions, as long as the total number of elements doesn't exceed 20 (when OPTION BASE 0 is in effect) or 12 (when OPTION BASE 1 is in effect).

If the right side of the equals sign is a parenthesized scalar expression, it is evaluated and each element of the receiving array is set to that value.

Example 2

```
MAT AR% = (4*10.02+8/3)
```

Every element in the array AR% is set equal to 43.

When the array is a character array, the character assigned may not exceed the defined maximum length (either explicit or implicit definition) of the elements of the character array. If there is an attempt to assign character data that exceeds this length, a string overflow occurs.

All array statements which result in assignment (with the exception of AIDX and DIDX) may have expressions on the right side of the equal sign that have the same array name as the one on the left. This is termed "in-position" replacement. Both of the following statements are permitted.

Example 3

```
100 MAT A = A + A
110 MAT A = INV(A)
```

If an error occurs, such as numeric overflow or string overflow, an exception is generated at the point of the error. This means that the array assignment may be partially complete (part of the array on the left side may have been changed). In order to allow recovery from such situations, IBM BASIC assures that the operands of the right side expression have not been altered when the exception occurs. If the array on the left appears as an operand on the right (including as an array element in a scalar expression), the right side is evaluated into a temporary array and then moved to the left side array.

For numeric MAT statements where the type (integer or decimal) of the array expression does not agree with that of the array to the left of the equal sign, the result of the expression is converted to the type of the array on the left. If necessary, rounding occurs prior to the assignment.

REDIMENSION CLAUSE: When the redimension clause is specified, this array is redimensioned dynamically; that is, the number of dimensions and the size in each dimension is changed to match the number of dimensions and the size of each dimension specified by the values in the redimension specification on the right.

Example 4

```
120 DIM A(100), B(3,4), C(4,3)
130 MAT A = B*C !A redimensioned to (3,3)
140 MAT A = C*B !A redimensioned to (4,4)
150 MAT A = A(2,2) !A redimensioned to (2,2)
```

When an array is redimensioned dynamically, the upper bounds are changed to match the size of its new value and the current lower bounds as defined by the OPTION BASE are retained. The new bounds need not be individually less than or equal to the bounds specified in the dimension statement for that array (or by the default dimensions if the array is not dimensioned explicitly), as long as the total number of elements for the redimensioned array does not exceed the total number of elements specified by the array's original dimensions. What this means is that for array B in the above example, you can change any one of the dimensions or even add dimensions, as long as the total number of elements doesn't exceed 20 (when OPTION BASE 0 is in effect) or 12 (when OPTION BASE 1 is in effect).

Array Assignment

Format

```
MAT arrayname = arrayname1 [(redimension)]
```

An element-by-element assignment is made to the array to the left of the equal sign.

If arrayname is a character string array, arrayname1 must be a character string array.

If arrayname is a numeric array, arrayname1 must be a numeric array. As the elements are assigned they are converted to the type of arrayname.

If the redimension specification is not given, the array to the left of the equal sign is dynamically redimensioned to the dimensions of the array on the right.

If the redimension specification is given, the array to the left of the equal sign is dynamically redimensioned to values specified in the redimension specification.

Example

```
100 OPTION BASE 1
110 DIM A(100), B(10,10)
120 MAT A=B
```

When the MAT statement is executed, array A assumes the dimensions and values of array B. However, if the following statement is executed instead:

```
120 MAT A=B (100)
```

Array A assumes the values of array B, but is still a one-dimensional array of 100 elements.

Scalar Assignment

Format

```
MAT arrayname = (expression) [(redimension)]
```

The value of the expression is assigned to each element of the array.

If arrayname is a character string array, the expression must be a character string expression.

If arrayname is a numeric array, the expression must be a numeric expression. The value of the expression is converted to the type of arrayname.

If the redimension specification is not given, the dimensions of the array are not changed.

If the redimension specification is given, the array is redimensioned to the values specified in the redimension specification.

MAT (Array Assignment) Statement

Example

```
200 MAT A = (PI/2)
```

This statements sets each element of the array A to the value PI/2.

Addition and Subtraction in Numeric Arrays

Format

```
MAT arrayname = arrayname1+arrayname2
                [(redimension)]

          or

MAT arrayname = arrayname1-arrayname2
                [(redimension)]
```

Each element of arrayname2 is added to (or subtracted from) the corresponding element of arrayname1 and the result is assigned to the corresponding element of arrayname.

Arrayname, arrayname1, and arrayname2 must all be numeric arrays. Mixed type operations are handled as described in "Mixed Type Numeric Expressions" on page 28.

Arrayname1 and arrayname2 must have the same number of dimensions and the same size in each dimension.

If the redimension specification is not given, arrayname is dynamically redimensioned to the dimensions of arrayname1.

If the redimension specification is given, arrayname is dynamically redimensioned to the values specified in the redimension specification.

Example

Assume each member of an array named ARAZ is to be given the sum of the corresponding members of arrays ARAY and ARAX. The value of ARAY(1) is to be added to the value of ARAX(1) and that sum stored in ARAZ(1), etc., until all of the values of ARAY and ARAX have been added together and stored in ARAZ. The source program coding will look like this:

```
 50 OPTION BASE 1
100 DIM ARAX(5),ARAY(5),ARAZ(5)
110 MAT ARAZ = ARAY + ARAX
```

and execution results are shown in Figure 29 on page 187.

If the function is changed to subtract, by changing line 110:

```
110 MAT ARAZ = ARAY - ARAX
```

the five values of ARAX are subtracted from the five values of ARAY and the differences stored in array ARAZ. Execution results are shown in Figure 29 on page 187.

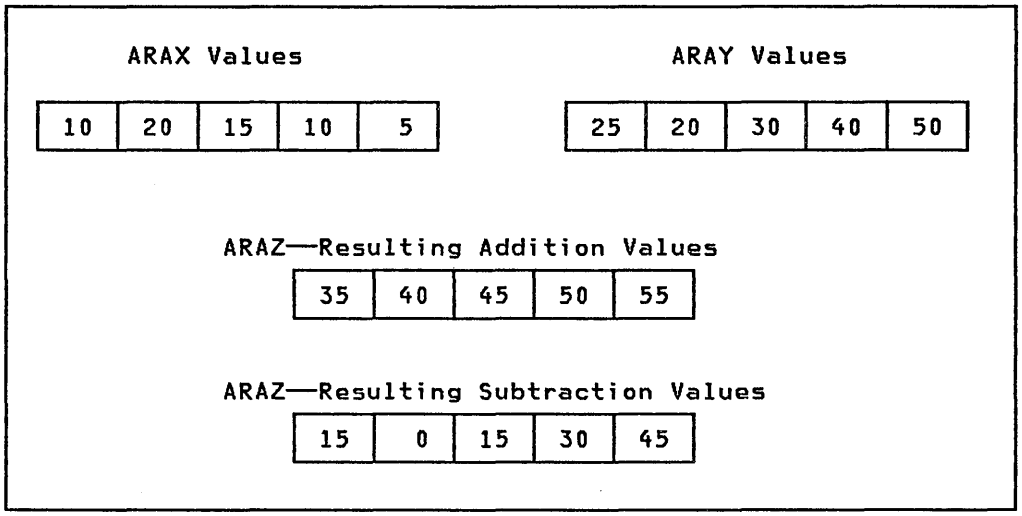


Figure 29. MAT Statement—Addition and Subtraction Example

Matrix Multiplication of Numeric Arrays

Format

```
MAT arrayname = arrayname1*arrayname2
                [(redimension)]
```

This statement performs the mathematical matrix multiplication of two numeric arrays and assigns the product to the third numeric array.

Each element of the result is the dot product of a row of the first array with a column of the second.

The two arrays (arrayname1 and arrayname2) must be two-dimensional. The number of columns of arrayname1 must be equal to the number of rows in arrayname 2.

Remember that the first subscript in a two-dimensional array indicates the number of rows, and the second the number of columns.

The result of the matrix multiplication is an array with the same number of rows as arrayname1 and the same number of columns as arrayname2.

If the redimension specification is given, after assigning the result to arrayname, arrayname is redimensioned to the values specified in the redimension specification.

Example

Assume ARAX and ARAY contain the values 2, 3, 4, 5, 6, 7 and 8, 9, 10, 11, 12, 13, respectively, and that you want place the results of matrix multiplication in ARAZ. The program coding would look like this:

```
 50 OPTION BASE 1
100 DIM ARAX (3,2), ARAY(2,3), ARAZ(3,3)
110 MAT ARAZ=ARAX*ARAY
```

When these statements are executed, the results are as shown in Figure 30.

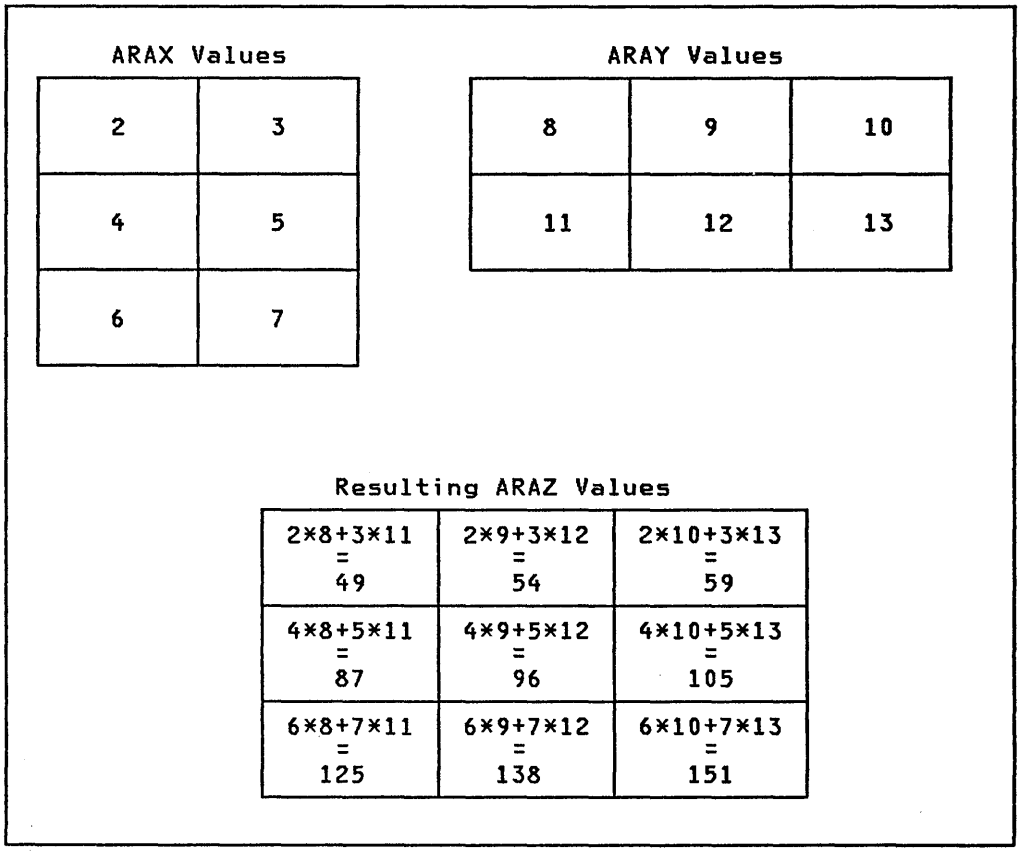


Figure 30. MAT Statement—Matrix Multiplication Example

The MAT statement in Figure 30 is equivalent to the following nested loops:

```
110 FOR I=1 TO 3
120 FOR J=1 TO 3
130 ARAZ(I,J)=0
140 FOR K=1 TO 2
150 ARAZ(I,J)=ARAZ(I,J)+ARAX(I,K)*ARAY(K,J)
160 NEXT K
170 NEXT J
180 NEXT I
```

Scalar Multiplication in Numeric Arrays

Format

```
MAT arrayname = (expression)*arrayname1
                [(redimension)]
```

Scalar multiplication is the process where each member of an array (arrayname1) is multiplied by the scalar result of an expression; that is, by the same number (the scalar multiplier). The results are stored in the array identified as arrayname.

Example

```
100 DIM ARAX (10,5),ARAY(14)
110 MAT ARAX = (2+2)*ARAY
```

In statement 110, 2+2 is evaluated, giving 4; the value of each member of ARAY is then multiplied by 4, and the product is assigned to the corresponding member of ARAX.

Array Concatenation of Character Arrays

Format

```
MAT arrayname = arrayname1 & arrayname2
                [(redimension)]
```

Each element of arrayname2 is concatenated to (joined together with) the corresponding element of arrayname1 and the result is assigned to the corresponding element of arrayname.

Arrayname, arrayname1, and arrayname2 must all be character arrays.

Arrayname1 and arrayname2 must have the same number of dimensions and the same size in each dimension.

If the redimension specification is not given, arrayname is dynamically redimensioned to the dimensions of arrayname1.

If the redimension specification is given, arrayname is dynamically redimensioned to the values specified in the redimension specification.

Example

Assume you have two arrays, ARA$ and ARB$, dimensioned 2 X 2, and that you want to concatenate their values together and place the result in ARC$. The program coding will look like this:

```
100 MAT ARC$ = ARA$ & ARB$
```

and the execution results are shown in Figure 31 on page 190.

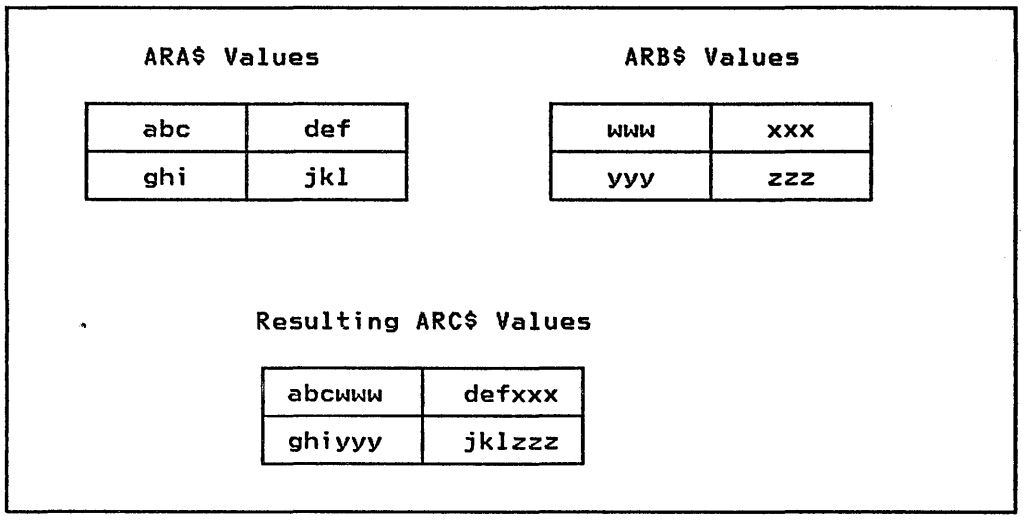


Figure 31. MAT Statement—Matrix Concatenation Example

Scalar Concatenation in Character Arrays

Format

```
MAT arrayname = (expression) & arraynamel
                [(redimension)]

          or

MAT arrayname = arraynamel & (expression)
                [(redimension)]
```

The value of expression is concatenated to (joined together with) each element of arraynamel and assigned to the corresponding element of arrayname.

If (expression) precedes arraynamel, the value of (expression) is concatenated on the left.

If (expression) follows arraynamel, the value of (expression) is concatenated on the right.

Arrayname and arraynamel must both be character arrays. Expression must be a character expression.

If the redimension specification is not given, arrayname is dynamically redimensioned to the dimensions of arraynamel.

If the redimension specification is given, arrayname is dynamically redimensioned to the values specified in the redimension specification.

Assume you have an array, ARA$, dimensioned 2 X 2, and a single 3-character variable DATA$, and you want to concatenate the value in DATA$ to the left of each element in ARA$ and place the result in ARC$. The program statement looks like this:

```
100 MAT ARC$ = (DATA$) & ARA$
```

and execution results are shown in Figure 32 on page 191.

However, if you want to concatenate the value in DATA$ to the right of each element in ARA$, the program statement looks like this:

```
100 MAT ARC$ = ARA$ & (DATA$)
```

and execution results are shown in Figure 32.

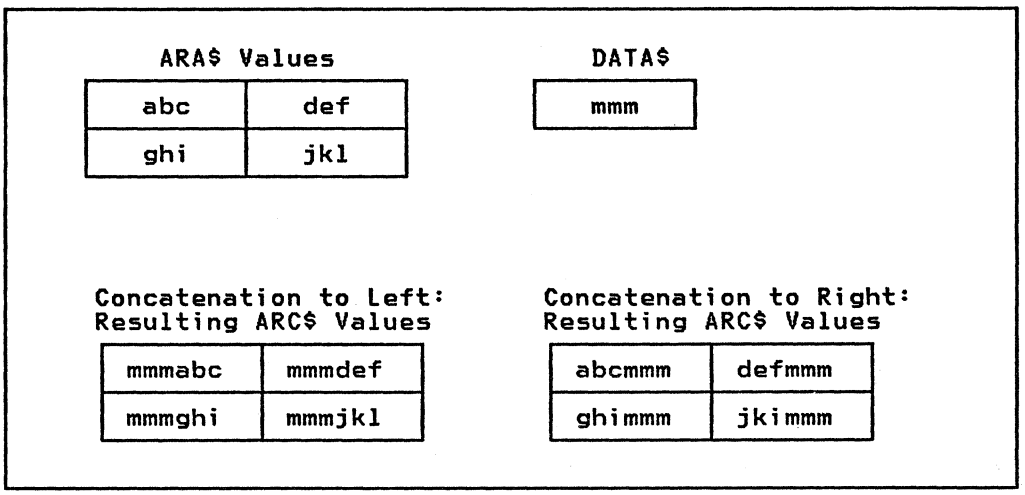


Figure 32. MAT Statement—Scalar Concatenation Example

If a character string being assigned to an element of ARC$ exceeds the maximum character length of the ARC$ element, a string overflow occurs.

Identity Array Function (IDN)

Format

```
MAT arrayname = [(expression)*]IDN
                [(redimension)]
```

The identity function, IDN, assigns the value of the expression (or one, if expression is not specified) to each diagonal array element (one whose subscripts are equal) and zero to all other elements.

Arrayname must be a numeric array. Expression must be a numeric expression.

If the redimension specification is not given, arrayname must be a two-dimensional array such that the number of rows equals the number of columns.

If the redimension specification is given, the redimension specification must specify a two-dimensional array such that the number of rows equals the number of columns. Arrayname is redimensioned to the values given in the redimension specification.

If a scalar-multiplier is used with the identity function, the value of the expression is assigned to each diagonal array member (one whose subscripts are equal), and assigns zero to all other array members.

Example

The following statements assign the value 1 to ARAX(1,1), ARAX(2,2), ARAX(3,3); all other array members are assigned the value 0.

```
 50 OPTION BASE 1
100 DIM ARAX(3,3)
110 MAT ARAX = IDN
```

Execution results are shown in Figure 33.

The following statements assign the value of 8 to AA(0,0), AA(1,1), AA(2,2) and AA(3,3); all other array members are assigned a value of zero.

```
100 DIM AA(3,3)
110 MAT AA = (8)*IDN
```

Execution results are shown in Figure 33.

Values in ARAX(3,3)

1	0	0
0	1	0
0	0	1

Values in AA(3,3)

8	0	0	0
0	8	0	0
0	0	8	0
0	0	0	8

Figure 33. MAT Statement—IDN Function Examples

Zero Array Function (ZER)

Format

```
MAT arrayname = ZER [(redimension)]
```

The ZER function sets the value of each element of the array to zero.

Arrayname must be a numeric array.

If redimensioning is not specified, the dimensions of arrayname are unchanged.

Example

The following statements apply the value 0 to all elements of ARAX:

```
100 DIM ARAX(3,3)
110 MAT ARAX=ZER
```

Execution results are shown in Figure 34 on page 193.

ARAX

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Figure 34. MAT Statement—ZER Function Example

Constant Array Function (CON)

Format

```
MAT arrayname = [(expression)*]
                CON [(redimension)]
```

The CON function sets the value of each element to the value of the expression (or to 1, if expression is not specified).

Arrayname must be a numeric array and expression must be a numeric expression.

If redimensioning is not specified, the dimensions of arrayname are unchanged.

Example

```
100 DIM ARAX (2,2)
110 MAT ARAX=CON
```

In this example, ARAX is set to all ones, by not specifying a numeric expression prior to CON.

By replacing line 110 with this assignment statement

```
110 MAT ARAX=(12)*CON
```

all of the values of ARAX become twelve.

Note: 110 MAT ARAX=(12)*CON yields the same results as 110 MAT ARAX=(12)

Null String Array Function (NUL$)

Format

```
MAT arrayname = NUL$ [(redimension)]
```

The NUL$ function sets the elements of a character array to null character strings.

Arrayname must be a character array.

MAT (Array Assignment) Statement

Example

```
100 DIM ARAA$ (2,2)
110 MAT ARAA$=NUL$
```

After the MAT statement is executed, ARAA$ is still a two-dimensional array, each element is a null character string. Note that this yields the same results as:

```
100 DIM ARAA$(2,2)
110 MAT ARAA$=("")
```

Inverse Array Function (INV)

Format

```
MAT arrayname = INV(arrayname1) [(redimension)]
```

The inverse function performs the matrix inverse of one square numeric array (a two-dimensional array with the same number of rows as columns) and assigns it to another array.

The inverse of an array is an array such that if the array and its inverse are multiplied, the result yields an identity matrix. Note the result of multiplying an array and the result of the INV function may not exactly equal the identity matrix due to roundoff and precision restrictions.

The array on the right (arrayname1) must be two dimensional and square.

Arrayname is dynamically redimensioned to the dimensions of arrayname1. If a redimension specification is given, after the assignment, arrayname is dynamically redimensioned to the value given in the redimension specification.

Not every matrix has an inverse. The inverse of a matrix exists if the DET function returns a value other than 0. The DET function can test for an inverse before inverting the array, if the array is specified as an argument to the DET function, thus checking for a value other than zero. (See "DET[(A)]" on page 39.)

Example

The following statements assign the inverse of array ARAK to array ARAJ:

```
100 OPTION BASE 1
110 DIM ARAJ(2,2),ARAK(2,2)
150 MAT ARAJ = INV(ARAK)
```

Execution results are shown in Figure 35.

ARAK Values

1	1
1	2

Resulting ARAJ Values

2	−1
−1	1

Figure 35. MAT Statement—INV Function Example

MAT (Array Assignment) Statement

Example

```
100 DIM ARAA$ (2,2)
110 MAT ARAA$=NUL$
```

After the MAT statement is executed, ARAA$ is still a two-dimensional array, each element is a null character string. Note that this yields the same results as:

```
100 DIM ARAA$(2,2)
110 MAT ARAA$=("")
```

Inverse Array Function (INV)

Format

```
MAT arrayname = INV(arrayname1) [(redimension)]
```

The inverse function performs the matrix inverse of one square numeric array (a two-dimensional array with the same number of rows as columns) and assigns it to another array.

The inverse of an array is an array such that if the array and its inverse are multiplied, the result yields an identity matrix. Note the result of multiplying an array and the result of the INV function may not exactly equal the identity matrix due to roundoff and precision restrictions.

The array on the right (arrayname1) must be two dimensional and square.

Arrayname is dynamically redimensioned to the dimensions of arrayname1. If a redimension specification is given, after the assignment, arrayname is dynamically redimensioned to the value given in the redimension specification.

Not every matrix has an inverse. The inverse of a matrix exists if the DET function returns a value other than 0. The DET function can test for an inverse before inverting the array, if the array is specified as an argument to the DET function, thus checking for a value other than zero. (See "DET[(A)]" on page 39.)

Example

The following statements assign the inverse of array ARAK to array ARAJ:

```
100 OPTION BASE 1
110 DIM ARAJ(2,2),ARAK(2,2)
150 MAT ARAJ = INV(ARAK)
```

Execution results are shown in Figure 35.

ARAK Values

1	1
1	2

Resulting ARAJ Values

2	-1
-1	1

Figure 35. MAT Statement—INV Function Example

INV accepts integer or decimal arrays as arguments but always produces decimal results.

If the argument of the INV function is singular (does not have an inverse), an exception occurs. If the argument for INV is not a square matrix, an exception occurs.

Transpose Array Function (TRN)

Format

```
MAT arrayname = TRN(arrayname1) [(redimension)]
```

The function TRN transposes an array. The values contained in column 1 of one array are transferred into row 1 of the other, the values in column 2 are transferred into row 2, etc.

Arrayname and arrayname1 must both be numeric arrays.

Arrayname1 must be a two-dimensional array.

Arrayname is redimensioned to be two dimensional with the number of rows equal to the number of columns in arrayname1 and with the number of columns equal to the number of rows in arrayname1.

After the assignment, if a redimension specification is given, arrayname is dynamically redimensioned to the value specified in the redimension specification.

Example

The following statements transpose the values of ARAX in ARAY.

```
100 OPTION BASE 1
110 DIM ARAX(3,4), ARAY(4,3)
120 MAT ARAY = TRN(ARAX)
```

Execution results are shown in Figure 36 on page 196.

If ARAX contained the values:

1	10	100	1000
2	20	200	2000
3	30	300	3000

ARAY would contain:

1	2	3
10	20	30
100	200	300
1000	2000	3000

Figure 36. MAT Statement—TRN function Example

Ascending Index Array Function (AIDX)

Format

```
MAT arrayname = AIDX(arrayname1)
                [(redimension)]
```

The AIDX function sorts an array in ascending sequence, and assigns the index (subscripts) of this sorted sequence to another array.

Arrayname must be a numeric array. Arrayname1 may be a numeric or a character array.

Arrayname1 must be a one-dimensional array (vector).

Arrayname is redimensioned to be a one-dimensional array with the same number of elements as arrayname1. After the assignment, if a redimension specification is given, arrayname is dynamically redimensioned to the values given in the redimension specification.

The array being indexed, arrayname1, is not changed.

The OPTION BASE setting affects the results of the AIDX function. Note that the subscripts of the indexed array are the results of the AIDX function; therefore OPTION BASE setting affects the results obtained.

Character arrays are indexed according to the collating sequence specified. Therefore, if arrayname1 is a character array, the OPTION COLLATE settings affect the results.

Example

The following statements index the elements of ARAA and assign them to ARAB in ascending sequence:

```
 90 OPTION BASE 1
100 DIM ARAA(4),ARAB(4)
110 READ MAT ARAA
120 DATA 31,13,46,20
130 MAT ARAB = AIDX(ARAA)
```

The created index lists the subscripts of the array elements in ascending order of the values contained within those elements, as follows: 2, 4, 1, 3, and then stores them in ARAB.

Now, to print the contents of ARAA in ascending order, the following loop can be specified:

```
150 FOR I=1 to 4
160 PRINT ARAA(ARAB(I))
170 NEXT I
```

generating

```
13
20
31
46
```

Descending Index (DIDX)

Format

```
MAT arrayname = DIDX(arrayname1)
                [(redimension)]
```

The DIDX function logically sorts every element in an array in descending order and assigns the index (subscripts) of this sorted sequence to another array. This is referred to as a descending index.

Arrayname must be a numeric array. Arrayname1 may be a numeric or a character array.

Arrayname1 must be a one-dimensional array (vector).

Arrayname is redimensioned to be a one-dimensioned array with the same number of elements as arrayname1.

After the assignment, if a redimension specification is given, arrayname is dynamically redimensioned to the values given in the redimension specification.

The array being indexed is not changed.

The OPTION BASE setting affects the results of the DIDX function. Note that the subscripts of the indexed array are the results of the DIDX function, therefore, OPTION BASE setting affects the results obtained.

Character arrays are indexed according to the collating sequence specified. Therefore, the OPTION COLLATE setting affects the results.

Example

The following statements index the elements of character array (ARA$) and assigns them to ARB% in descending sequence (default OPTION BASE 0 is in effect):

```
100 DIM ARA$(4),ARB%(4)
110 MAT READ ARA$
120 DATA EASY, CHARLIE, ABLE, DOG, BAKER
130 MAT ARB% = DIDX(ARA$)
```

The index created lists the subscripts of the array elements in descending order of the values contained within those elements (0, 3, 1, 4, 2) and stores them in ARB%.

To print the contents of ARB$ in descending order, the following loop may be used:

```
150 FOR I = 0 to 4
160 PRINT ARA$ (ARB%(I))
170 NEXT I
```

generating

```
EASY
DOG
CHARLIE
BAKER
ABLE
```

Sort Array Functions (ASORT, DSORT)

Format

```
MAT arrayname = ASORT(arrayname1)
                [(redimension)]

      or

MAT arrayname = DSORT(arrayname1)
                [(redimension)]
```

The ASORT function sorts character or numeric arrays in ascending sequence. The DSORT function sorts character or numeric arrays in descending sequence. The array being sorted is arrayname1.

Character arrays are sorted based on the collate option selected. See "OPTION Statement" on page 211.

If arrayname is a character array, arrayname1 must be a character array. If arrayname is a numeric array, arrayname1 must be a numeric array.

Arrayname is redimensioned to the dimensions of arrayname1. The sorted values are then stored into arrayname such that the rightmost subscripts vary most rapidly.

After the assignment, if a redimension specification is given, arrayname is dynamically redimensioned to the values given in the redimension specification.

Example

```
100 DIM ASC$(3,3),ASB$(3,3)
110 MAT ASB$ = ASORT(ASC$)
120 DIM DEC$ (3,3), DEB$ (3,3)
130 MAT DEB$ = DSORT(DEC$)
```

OPTION COLLATE NATIVE is in effect, therefore, the members of array ASC$ are sorted according to the EBCDIC collating sequence and stored in ascending order in array ASB$.

OPTION COLLATE NATIVE is in effect, therefore, the members of array DEC$ are sorted according to the EBCDIC collating sequence and stored in descending order in DEB$.

Note: The EBCDIC collating sequence can be changed to the ASCII collating sequence if OPTION COLLATE STANDARD is specified.

Example

```
100 DIM AA(100),BB(100)
110 MAT AA = ASORT(BB)
```

The members of array BB are sorted in ascending numeric sequence and stored in ascending order in array AA.

For numeric arguments, the same type argument is returned (integer, decimal) and the result is then converted to the type of array to the left of the equal sign.

Immediate Execution

All forms of the MAT statement may be used in the immediate mode.

Immediate and program variables may be used in expressions, but scalar expressions cannot refer to functions defined in a program. Intrinsic functions can be used.

"Immediate Statements" on page 260 gives additional information.

NEXT Statement

NEXT STATEMENT

The NEXT statement is the end delimiter of a FOR loop.

Format

```
NEXT variable
```

Where:

variable
is a numeric variable, the "control variable" used in the corresponding FOR statement.

Description

The FOR and NEXT statements enclose a set of statements which are executed zero or more times depending on the evaluation of the expressions associated with the FOR statement. The NEXT statement must follow its associated FOR statement in line number sequence.

When the NEXT statement is executed, the control variable is incremented and compared to the final value specified in the FOR.

For complete details, see "Loop Control Statements" on page 62 and "FOR Statement" on page 127.

ON GO TO/GOSUB STATEMENT

The ON GOTO and ON GOSUB statements conditionally transfer control to one of a group of statements. ON GOSUB also saves the return location.

Format

```
ON numeric expression {GOTO|GOSUB} s[,s]...
            [NONE s|ELSE statement]
```

Where:

numeric expression
can be any numeric expression as described in "Numeric Expressions" on page 25.

s
is a line number or line label.

statement
is an imperative statement.

Description

The ON GOTO and ON GOSUB statements conditionally transfer control to one of a series of statements, depending on the value of a numeric expression.

(GOTO may also be spelled GO TO; GOSUB may also be spelled GO SUB.)

The rounded numeric expression is evaluated and its value determines the element of the line number list to which the GOTO or GOSUB statement branches.

Example

```
100 ON ABLE GOTO 120, 130, 140
```

If ABLE equals 1, the program branches to 120, if ABLE equals 2, the program branches to 130, if ABLE equals 3, the program branches to 140.

If the rounded value of the expression is not represented in the list, an error condition occurs which can be handled by either the NONE or ELSE clause. If neither the NONE nor the ELSE clause is present, and the value of the expression is not represented in the list, an exception occurs.

The NONE clause allows the program to branch to a predetermined line number or line label when the value of the expression is not represented in the list.

Example

```
100 ON I% GOTO 120,130 NONE 980
```

If I% equals any value less than one or greater than two, the program branches to statement 980.

If the line branched to is a nonexecutable statement, control is passed to the first executable statement following the specified statement.

Transfer to a nonexistent line number results in an exception.

If ELSE is specified and the value of the expression is not represented in the list, the imperative statement is executed. That statement may be any of those shown in Figure 25 on page 149.

Example

```
100 ON A+B GOTO 120,130 ELSE PRINT "NO TRANSFER"
110 X = X+Y
```

If A+B equals any value less than one or greater than two, the program executes the PRINT statement, and processing continues with the next statement (110) in sequence.

ON GOTO: unconditionally branches to the line number or line label in the list.

ON GOSUB: passes control in the same manner as the ON GOTO statement, with one exception. The line numbers represent the first statement of a subroutine and, as with any other subroutine process, when it is complete (by execution of a RETURN statement indicating the end of the subroutine) the statement immediately following the ON GOSUB statement is executed.

Execution of a RETURN statement is the normal completion of an ON GOSUB statement, in which case program execution is returned to the statement following the ON GOSUB. However, as described for the GOSUB and RETURN statements, termination of a program unit or multiline function (execution of a SUBEXIT or FNEND statement) deactivates all ON GOSUBs associated with that program unit or function. See "Subroutine Control Statements" on page 61.

Example

```
100 INPUT "ENTER DESIRED ACTION NUMBER": ACTION%
110 ON ACTION% GOSUB DEBIT, CREDIT, CURRENT_BALANCE&
& NONE 500
```

When the ON GOSUB statement is executed, control is transferred as follows:

If ACTION% is:	Control is transferred to:
1	DEBIT
2	CREDIT
3	CURRENT_BALANCE

If ACTION% contains any other value, control is transferred to line 500.

ON CONDITION STATEMENT

The ON condition statement indicates the action to be taken when an exception occurs.

Format

```
ON condition action
```

Where:

condition
is one of the following:

ATTN	CONV	ENDPAGE
ERROR	OFLOW	PAGEOFLOW
SKEY	SOFLOW	UFLOW
ZDIV		

action
is stated as one of the three options:

```
IGNORE
GOTO s
SYSTEM
```

s
is a line number or line label.

Description

The ON Condition statement is an executable statement that, when executed, establishes what action is to be taken if the program subsequently generates an exception. Exceptions are grouped according to which of the following conditions they represent:

ATTN The "attention" interrupt from a terminal, or its equivalent has occurred.

CONV An error has occurred during an input/output operation. The error can be a numeric data conversion error, or can be due to mismatched record descriptions.

ENDPAGE A PRINT or PRINT File statement has attempted to start a new line beyond the limits specified for the current page. See "MARGIN Statement" on page 178.

ERROR Is a generalized exception; it applies to any error condition not specifically stated in this list.

OFLOW The condition of numeric overflow has occurred. This happens when a computed value exceeds the allowed range.

PAGEOFLOW The same as ENDPAGE.

SKEY One of the PF keys on a 327X terminal, or equivalent, has been pressed. An SKEY exception can only occur during the execution of an INPUT, LINE INPUT or INPUT FIELDS statement when the user responds to the request for input with a PF key rather than the ENTER key. When an SKEY exception occurs and causes a transfer of control, the function KEYNUM returns the number of the PF key pressed.

SOFLOW A string overflow condition has occurred; that is, character data has been moved into a field that is too small to contain it.

UFLOW The condition of numeric underflow has occurred. This occurs when the computed value is smaller than the smallest decimal value allowed.

ZDIV The condition of division by zero of numeric data has occurred. Zero raised to a negative power also produces an exception which is classed as a ZDIV exception.

ON CONDITION RESPONSES: You may pick one of three actions to occur when the specified condition happens, as follows:

IGNORE Allows processing to continue normally

GOTO s Causes the indicated transfer of control to a line label or line number.

SYSTEM Permits a predetermined system-controlled response to the condition

EXIT clauses referring to the same conditions as those in the ON override the ON-Condition action.

Whenever an exception causes a transfer of control, the exception code is available by using the function ERR. The line number of the statement where the exception occurred is available by using the LINE intrinsic function. In addition, when an SKEY exception occurs, the function KEYNUM returns the number of the PF key which was pressed.

Figure 37 indicates which of the actions may be used with a particular condition and what they mean.

ON	IGNORE	GO TO s	SYSTEM
ATTN	No user message Continue	No user message Transfer control	Batch execution: ignore Interactive execution: stop at the next statement.
CONV	Not allowed	No user message No data transfer Transfer control	User message Error Stop
ENDPAGE or PAGEOFLOW	No user message Continue	No user message Transfer control upon completion of the input/output statement causing the ENDPAGE.	Generate top of form (if the device or file allows it) and print blank lines for top margin See "MARGIN Statement" on page 178.
ERROR	Not allowed	No user message No data transfer Transfer control	User message Error Stop

Figure 37 (Part 1 of 2). ON Conditions—Processor Actions

ON	IGNORE	GO TO s	SYSTEM
OFLOW	No user message Replace with signed INF See "INF" on page 42 Continue	No user message No data transfer Transfer control	User Message Replace with signed INF See "INF" on page 42 Continue
SKEY	No user message Continue	No user message Transfer control	No user message Continue
SOFLOW	No user message Excess data is truncated on the right Continue	No user message No data transfer Transfer Control	User message Error Stop
UFLOW	No user message Replace with zero Continue	No user message No data transfer Transfer control	User message Replace with zero Continue
ZDIV	No user message Replace with signed INF See "INF" on page 42 Continue	No user message No data transfer Transfer Control	User message Replace with signed INF See "INF" on page 42 Continue

Figure 37 (Part 2 of 2). ON Conditions—Processor Actions

OPEN STATEMENT

The OPEN statement activates a file and specifies access conditions.

Format

```
OPEN #fileref:[NAME]fileid [file attributes] [err]
```

Where:

fileref
is a numeric expression which when evaluated and rounded, must be a positive integer within the range 0 to 255.

fileid
is a character expression, the value of which contains all of the information necessary to define the file to the system. This field is system dependent. See IBM BASIC Application Programming: System Services for valid entries for your system.

If the file is a display output file which is to include a carriage control character at the beginning of each record, the following field must appear as the final character values in the character expression:

```
,DEVICE PRINTER
```

If the file is a display output file to be listed on a 3800 printer, the following field may appear as the final character values in the character expression:

```
,DEVICE 3800
```

which specifies font control for a 3800 device. It specifies that a second control character (the first is for carriage control) is necessary for font control on output, when using the PRINT File statement for the 3800 device. DEVICE 3800 and DEVICE PRINTER may both appear, even though DEVICE PRINTER is then redundant.

Note: Font in this context refers to a collection of type, all of one size and style, for a printer.

file attributes
specify, in any order, file access, type, organization, position, and record type. The options are:

```
ACCESS
ORGANIZATION
POINTER
RECORDS
TYPE
```

These options are described below.

err
can be:

```
EXIT line-ref

   or

IOERR line-ref
```

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and IOERR are mutually exclusive.

Description

The OPEN statement makes an external file available for processing by the program.

FILEREF: The fileref clause supplies a file reference number by which the file is referenced in all subsequent input/output statements for that file; a fileref of 0 refers to the terminal. An attempt to access a file (except file 0) which has not been opened results in an exception. File 0 is always open; a request to open it is ignored.

FILE ATTRIBUTES: The operating conditions for the file are defined by the file attributes clauses, which are defined as follows:

1. File Access: Specifies what input/output operations are allowed.

```
Format
    [,[ACCESS] {OUTIN|INPUT|OUTPUT}]
```

   a. INPUT specifies that only read operations will be performed on this file while this OPEN is in effect. No replacement or deletion is permitted.

      For internal format files (including stream files and DISPLAY files), INPUT is the default if no access is specified.

   b. OUTPUT specifies that only write operations will be performed on this file while this OPEN is in effect.

   c. OUTIN specifies that both read and write operations are valid on the file while this OPEN is in effect. Sequential files may be extended, but not shortened.

   For native format files, OUTIN is the default if no access is specified.

2. File Type: Specifies the appearance of the records in the file.

```
Format
    [,[TYPE] {NATIVE|DISPLAY|INTERNAL}]
```

   a. DISPLAY specifies the file is to be written in the same format as the data that would have been displayed on a print output device. That is, each record is a single character string, consisting of the edited values from the data list, positioned according to the

specifications. The string is terminated by an end-of-record.

On output, a carriage control character is prefixed to each record, if DEVICE PRINTER is specified in the file-id string, and, if DEVICE 3800 is specified in the file-id string, a font control character is also prefixed after the carriage control character.

DISPLAY is the default if no file type is specified.

b. INTERNAL specifies that each record of an internal file is to be written as a sequence of numeric and string values. These files are written in internal binary format, each value preceded by a type byte. Internal files cannot be edited.

c. NATIVE specifies that the contents of each record are not self-defining. The program will format them.

3. File Organization: Specifies the method by which data is arranged.

Format

```
[,[ORGANIZATION] {SEQUENTIAL|RELATIVE|KEYED|STREAM}]
```

a. SEQUENTIAL specifies that the file can only be accessed sequentially.

If the organization is not specified, SEQUENTIAL is assumed.

b. RELATIVE specifies that the file can be accessed through reference to relative record numbers of records within that file.

c. KEYED specifies that the file can be accessed through reference to keys which exist within each record in that file.

d. STREAM specifies that the file is a sequential file, each of whose records contains a single character string or numeric value.

Only certain combinations of type and organization are allowed as shown in Figure 38.

	SEQUENTIAL	RELATIVE	KEYED	STREAM
DISPLAY	X			
INTERNAL	X			X
NATIVE	X	X	X	

Figure 38. Allowable Combinations of File Type and File Organization

4. File Pointer: Specifies whether or not the file should be opened so that data may be accessed from the beginning or added to the end.

```
Format

    [,[POINTER] {BEGIN|APPEND|END}]
```

For sequential files:

a. BEGIN specifies that the file will be opened at its beginning.

   If the pointer clause is omitted, BEGIN is the default for input files.

b. APPEND/END specifies that the file will be opened at a position following the last record in the file.

   If the pointer clause is omitted, END is the default for OUTPUT or OUTIN files.

For relative and keyed files:

a. BEGIN specifies that the file will be opened at its beginning.

b. APPEND/END have no meaning.

   Note: Opening an existing file with OUTPUT and BEGIN has the same effect as a SCRATCH statement. That is, all data in the file is lost and the file is ready to be created.

5. File Record Type: Specifies the length attribute of individual records in the data file. The attributes are:

```
Format

    [,[RECORDS]{VARIABLE[rec-length]|FIXED[rec-length]}]
```

a. Rec-length is a numeric expression, specifying the actual or maximum length of each record in the file.

b. VARIABLE specifies that records of different lengths may appear on the data file. If not stated, VARIABLE is the default value.

   The lengths of all records in a file can be no longer than that specified by the rounded integer value of the rec-length in the record type in the OPEN statement opening the file. The value of the rec-length must be in the range 0 to 32756. If no rec-length is specified (which includes the case where the entire RECORDS clause is omitted), the default maximum record size depends upon the file type:

DISPLAY	133
INTERNAL	255
NATIVE	255

   "VARIABLE 0" results in a maximum record size of 32756 (that is, VARIABLE 0 is treated the same as "VARIABLE 32756"

If the record length is omitted and if the file already exists, for standard system files the record length is set to the maximum record length presently existing in the file. Therefore, to write records longer than the longest record existing in the file, the new maximum record length must be specified in the OPEN statement.

For VSAM files under the same conditions as above, the record length is set to the maximum record length defined for the file and records longer than this cannot be written.

c. FIXED specifies that each record in the file has the same length as every other record in that file.

The rec-length specifies the actual length of all records in the file. The rounded integer value of rec-length must be in the range 1 to the maximum for the system. If no rec-length is specified, the default fixed record size depends upon the file type:

DISPLAY	133
INTERNAL	255
NATIVE	255

For fixed length records, if the file already exists and a record length is specified, it must match the record length in the file. If the record length is not specified, the record length existing in the file is used.

ERROR CONDITIONS: An attempt to open a file which has already been opened results in an error condition. Attempting to open a file with an invalid file reference number results in an error condition. These errors can be recovered from, if either the IOERR clause is provided, or an EXIT err clause is to refer to an EXIT statement which contains an IOERR clause.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 OPEN #2:    FILEA$,ACCESS INPUT,TYPE NATIVE,&
  & ORGANIZATION SEQUENTIAL,POINTER&
  & BEGIN,RECORDS FIXED IOERR 750
```

The above example opens a native sequential file containing fixed length records whose length is 255 characters. The file is positioned at its beginning and can be accessed for input only. If the open is not successful, control passes to line number 750.

```
200 OPEN #100: "PRINT1,DEVICE 3800",OUTPUT
```

This example opens SEQUENTIAL file 100 as a DISPLAY OUTPUT file, positioned at the END of the file. The RECORDS in the file are a maximum 133 characters long, and are VARIABLE in length. Font control is requested through DEVICE 3800. The following options are supplied by default:

```
SEQUENTIAL,DISPLAY,END,VARIABLE 133.
```

For examples of keyed and relative files, see IBM BASIC Application Programming: System Services.

OPTION STATEMENT

The OPTION statement permits the selection of a variety of options that can be applied to a program.

Format

```
OPTION option[, option]...
```

Where:

option
is one of the following keyword phrases:

```
BASE{0|1}
COLLATE{NATIVE|STANDARD}
INVP
{SPREC|LPREC}
PRTZO nn
RD nn
FLAG{I|W|E|S}
{FIPS|NOFIPS}
```

All the keyword phrases are discussed below.

Description

OPTION statements are used to define certain actions to be taken when language or data is encountered, either during the compilation of a program or the execution of the program. Options may also be stated on a COMPILE or RUN command; however, they are overridden by any options explicitly stated in an OPTION statement within the program.

Each option has a default action. Check with your system administrator for the defaults in effect for your organization.

The OPTION statement is a nonexecutable statement that can be placed anywhere in a program unit and which affects the entire program unit in which it is specified.

Options may appear in any order, in one or more OPTION statements. However, any given option may not be redefined within the same program unit. For example, once an option has been used to set the collating sequence to STANDARD it cannot be reset to NATIVE in the same program unit.

If the same option is declared more than once in a program unit, even if the declarations are redundant, an error message is printed.

The scope of an option declared by an OPTION statement is the containing program unit. Options for a subprogram become active when the subprogram is entered; the options for the calling program are reactivated when the subprogram is exited.

When a main program terminates normally or abnormally in the interactive environment, the program's options remain in effect until the program is edited or you explicitly reset the options (for example through an immediate OPTION statement). When execution is suspended at a breakpoint, the options remain those of the interrupted program unit.

Immediate mode options may be set with immediate OPTION statements.

The options available are:

BASE {0|1}
This option specifies whether or not array dimensions include elements corresponding to subscripts with a value of zero. This definition applies whether or not a DIM statement was used.

In the absence of an OPTION BASE specification, BASE 0 applies.

Arrays may be passed as parameters between subprograms having different bases, but subscripts obey the base of the program unit containing them. Because of the possible confusion different bases could cause, you should usually use the same base in all related program units.

Example

```
Main program:    110 OPTION BASE 0
                 120 DIM A(5,10)
                 130 A(0,0)=1
                 140 A(5,10)=2
                 150 CALL S1(A(,))
                 160 END
Subprogram:      200 SUB S1(D(,))
                 210 OPTION BASE 1
                 220 PRINT D(1,1)+D(6,11)
                 230 END SUB
```

The PRINT statement will print the value 3.

COLLATE {NATIVE|STANDARD}
This option specifies the collating sequence to be used for the comparison and conversion of character data.

If OPTION COLLATE NATIVE is in effect, the collating sequence is Extended Binary Coded Decimal (EBCDIC).

If OPTION COLLATE STANDARD is in effect, the collating sequence is the American National Standard Code for Information Interchange (ASCII).

If neither is specified, OPTION COLLATE NATIVE is the default.

Character data is always represented in EBCDIC. OPTION COLLATE only affects the comparison of character strings (relational expressions, ASORT, DSORT, AIDX, DIDX) and the intrinsic functions CHR$ and ORD.

(The EBCDIC and ASCII collating sequences are listed in "Appendix B. Character Set Collating Sequences" on page 327.)

INVP
INVP (inverted print edit facility) specifies that numeric values are printed interchanging the usage of the period (decimal point) and the comma, in order to print in the normal European format.

When this option is specified, a comma is printed as the decimal point in a numeric value in place of the period, and a period is printed in place of the comma when used for separating triples of a numeric value.

This interchange of period and comma applies to all output resulting from the execution of a print-type statement, PRINT or PRINT USING. This includes the commas/periods within an IMAGE statement and within a PIC in a FORM statement.

If INVP is not present, the standard U.S. specification for printing of commas and periods (decimal points) applies to print-type statement output.

Example

Without INVP	With INVP
123,456.78	123.456,78

The INVP option has no effect on the format in which data is present in the program or on the format in which data must be entered in response to an INPUT statement.

{SPREC|LPREC}
This option specifies the maximum number of significant decimal digits to be printed by the PRINT statement (without the USING clause) when printing decimal values.

SPREC specifies "short" precision of 6 digits.

LPREC specifies "long" precision of 12 digits.

PRTZO nn
This option specifies the width of zones to be used when printing. Unless otherwise stated, the width of each print zone is wide enough to permit printing of explicit point scaled data items. (See "PRINT Statement" on page 217.)

This default may be overridden by the use of OPTION PRTZO nn, where nn defines the print zone width. The value assigned to nn must be in the range:

nn >= the minimum as described above.

nn <= the minimum of 255 or the difference between the right and left margins.

Note: If nn is outside the width range, an error occurs.

The default value of nn is a parameter supplied during installation. As distributed, the default is 20.

RD nn
This option specifies the number of rounded decimal digits to be displayed when a PRINT statement (not a PRINT USING statement) is executed, in lieu of the default action of suppressing trailing zeros.

nn is in the range of (0 <= nn <= 12). If nn is outside that range, an error occurs. A decimal value to be printed is converted and rounded if necessary to nn digits to the right of the decimal point. Thus if RD 03 is specified, 4.5678 is printed as 4.568, and if RD 5 is specified, 4.5678 is printed as 4.56780.

FLAG ({I|W|E|S})
This option determines the level of error messages reported. Control of the error checking level also exists as an option to the processor, but OPTION FLAG overrides those supplied when the processor is invoked.

The levels (in increasing order of severity) are:

I Informative messages

W Warning messages

E Error messages

S Severe error messages

The OPTION FLAG statement specifies that only errors of levels higher than or equal to the indicated level are to be reported.

{FIPS|NOFIPS}
This option indicates whether or not the processor should produce a warning diagnostic for any statement which does not conform to the FIPS BASIC syntax. Federal Information Processing Standard (FIPS) BASIC is defined in the publication Minimal BASIC, FIPS PUB 68. Any program written to conform to FIPS BASIC must conform to the BASIC language defined in that publication.

OPTION FIPS is negated by OPTION FLAG(W|E|S), which direct the system to withhold the display of informational diagnostic messages (which is what the FIPS messages are).

Immediate Execution

The OPTION statement may be used in immediate mode with all of the parameters allowed in a program. However, immediate options obey different rules.

FIPS/NOFIPS AND FLAG OPTIONS: FIPS/NOFIPS and FLAG options are treated differently in immediate mode than they are when used in a program. In immediate mode, they temporarily change the state of interactive IBM BASIC so that it produces the indicated error messages. BASIC remains in this state until:

1. A contradictory immediate OPTION statement is executed.
2. An INITIALIZE command is executed.

For example,

```
OPTION FIPS
```

causes checking of all statements entered from the terminal and by LOAD or MERGE commands for deviations from the FIPS BASIC Standard.

When used in a program these options control the error messages produced when the program is compiled, either with the COMPILE command or as a batch compilation.

OTHER OPTIONS: All other immediate options (BASE, COLLATE, INVP, LPREC/SPREC, PRTZO, RD) have their normal meanings with the following restrictions:

1. The duration of these options is the same as immediate variables. They last until:
   - The workspace is edited
   - The next RUN command is executed
   - The next COMPILE command is executed
   - The next DROP (all) command is executed
2. Immediate options cannot be entered while at a breakpoint. Immediate options must agree with the options being used by the current program unit. Thus they are the same as the options which are implicitly in effect at that breakpoint.

3. OPTION BASE cannot contradict the base (0 or 1) of any existing immediate arrays (immediate arrays within scope when the OPTION BASE immediate statement is entered) (scope is defined in "Variables and Arrays and Immediate Statements" on page 261). If you want to change the base, you must DROP existing arrays.

"Immediate Statements" on page 260 gives additional information.

PAUSE Statement

PAUSE STATEMENT

In the interactive mode, the PAUSE statement halts execution of the program in which it appears. The statement is ignored during batch mode operation.

Format

```
PAUSE [pause-message]
```

Where:

pause-message
is an optional character expression.

Description

When the PAUSE statement is processed, program execution is halted.

The pause message, when specified, is displayed just prior to program interruption. If the pause message is omitted, the following message displays automatically:

```
PAUSE AT LINE line-number
```

and displaying the program line where execution stopped.

Example

```
220 PAUSE "COMPARE FAILED"
```

When statement 220 is executed, the program halts and the following message is displayed at the user's terminal:

```
COMPARE FAILED
```

To resume execution of the interrupted program, the user enters either a null line or the GO command.

PRINT STATEMENT

The PRINT statement displays data at the terminal.

Format

```
[MAT] PRINT [USING line-ref:]
            [output-list] [err,[err]...]
```

Where:

line-ref
is the line number or line label of an IMAGE or FORM statement, or a character expression which contains an IMAGE or FORM specification.

output-list
is a list of constants, variables, array elements, expressions (numeric and character), and array names (array names can only appear prefaced with the keyword MAT, either at the beginning of the statement or immediately preceding each array name).

List elements are separated by commas or semicolons. The keywords TAB (followed by a numeric expression in parentheses), PAGE, and NEWPAGE may be used in the list.

If the output list is omitted, a blank line is printed.

err
is one of the following:

EXIT line-ref

CONV line-ref

IOERR line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and the other err clauses are mutually exclusive.

The colon after line-ref may be omitted if it would be the last nonblank character on the line.

Example

```
150 PRINT USING 180:
```

and

```
150 PRINT USING 180
```

are equivalent.

PRINT Statement

General Description

The PRINT statement is used in three different ways to display data at the terminal:

- PRINT with no USING clause
- PRINT with the USING IMAGE clause
- PRINT with the USING FORM clause

All three formats deal with the concepts of the limits of the output line, the print zone, and separators.

OUTPUT LINE: The output line is first limited to the size of a line as defined by the system, and may be further limited by the left and right margins of the line.

The MARGIN statement can be used to specify these values, and, in effect, determine where output can begin and end on a line. (See "MARGIN Statement" on page 178.)

PRINT ZONE: The print zone is the number of positions allocated for printing data items; this is an installation parameter. As distributed by IBM, the default value is 20, a value sufficiently large to allow printing of floating point scaled data items in long precision format.

A different print zone value may be assigned by an OPTION PRTZO statement.

The print zone is constant throughout a program unit and must not be less than 13 for short precision or 19 for long precision. (It should be noted that character items are not limited to 20 positions; if larger, they will extend into new print zones).

SEPARATORS: The separators are the comma (,) and semicolon (;). The items of an output-list must be separated by commas or semicolons; the last item may be followed by a comma or a semicolon.

In general, a comma indicates that the current print position should be advanced to the next print zone.

If a comma appears when the print position is in the last print zone on a line, an end of line is generated.

In general, a semicolon indicates that the next printed value will appear in the position immediately following the last printed value.

Specific usages of the comma and semicolon are explained under each of the three PRINT formats below. They are summarized in Figure 39 on page 219.

MAT KEYWORD: The MAT keyword preceding the PRINT keyword specifies that the output-list consists only of arrays; the MAT keyword is then unnecessary in the output-list. See "Input/Output Lists" on page 70 for more information.

ERROR CONDITIONS: All three formats of the PRINT statement may employ the err clause to process CONV, SOFLOW, and IOERR conditions.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

1. PRINT without IMAGE or FORM

CHARACTER	TRAILING	IMBEDDED
,	same line, next print zone	same line, next print zone
;	same line, next character	same line, next character
no , or ;	next line	error

2. PRINT with IMAGE or FORM where USING clause is exhausted (Reuse IMAGE/FORM)

CHARACTER	TRAILING	IMBEDDED
,	same line, next print zone	new line
;	same line, next position	same line, next position
no , or ;	next line	error

3. PRINT with IMAGE or FORM where USING clause is not exhausted.

CHARACTER	TRAILING	IMBEDDED
,	same line, next print zone	same line, next position
;	same line, next position	same line, next position
no , or ;	next line	error

Figure 39. PRINT Statement—Comma and Semicolon Separator Usage

PRINT without USING Clause

When the USING clause is not specified in the PRINT statement, printed output is under control of the following factors:

- The line size dictated by the terminal
- The current settings of the margins
- The width of a print zone
- The separators in the output list
- The content and precision of numeric data
- The length of character data

The current setting of the margins can be controlled by the MARGIN statement. See "MARGIN Statement" on page 178.

The width of the print zone can be controlled by the OPTION PRTZO statement. See "OPTION Statement" on page 211.

SEPARATORS: When an imbedded or trailing comma appears in the output-list, the current line position is advanced to the next print zone.

When an imbedded or trailing semicolon appears, the next printed value appears immediately following the last printed value.

If neither appears, new data is displayed on the next line.

PRINT ZONE: The current length of a character string determines how many characters will print.

All numeric values are displayed in one of three forms:

```
implicit unscaled      sd...d
explicit unscaled      sd...drd...d
explicit scaled        sd...drd...dEsdd
```

where:

d is a decimal digit

r is a decimal point

s is an optional sign

E is the character E

Which of the three forms is used for a numeric value depends in part on the value w assigned by the OPTION statement:

- w is 6 if OPTION SPREC is in effect
- w is 12 if OPTION LPREC is in effect

Each number which can be represented exactly as an integer with w or fewer decimal digits is displayed using the implicit unscaled representation.

All other numbers are displayed in one of the two explicit forms:

- Numbers which can be represented with w or fewer digits in the explicit unscaled form no less accurately than they could be in the explicit scaled forms are displayed in the unscaled form.
- Numbers which cannot be represented with w or fewer digits in the explicit unscaled form as accurately as they can be in the explicit scaled forms are displayed in the scaled forms.

A printed numeric value is always separated from the next value on the line by a space, regardless of the separators in the output-list. If the number is positive, a space is printed in the first position; if the number is negative, a minus sign is printed in the first position. A plus sign or minus sign, as appropriate, is always printed before the E in explicit scaled form.

Data displayed in the last print zone of a line causes an advance to the next line.

If a character data item will not fit on the current line and the character data item is not the first item in the current line, the character data item is printed at the beginning of the next line.

If a character data item is the first item in the current line and the character data item is longer than the line, it is split into line length portions and printed on successive lines until the total length is printed.

The TAB clause sets the columnar position of the current line, prior to printing the next item. The numeric expression specified by the TAB is first evaluated and rounded to the nearest integer

n. If n is less than 1, a warning message is produced, and n defaults to 1.

If n is not less than one, the columnar position is set to the value:

```
left + MOD (n-1, right-left +1)
```

where left and right are the left and right margins, and MOD is the MOD intrinsic function. If n specifies a position prior to the current position in the line, the current line is written and n sets the position in the next line. This has the following "wraparound" effect, assuming that left=1 and right=80:

n	line position
1	1
10	10
80	80
81	1

The NEWPAGE or PAGE clause clears a terminal screen, or restores a hard copy terminal to a new page, and then resets the current print zone to the leftmost print zone. The action of the NEWPAGE clause occurs at the point in the output list where the keyword appears; therefore, if used other than as the first item in an output list, NEWPAGE clears the display of the previous items. For example:

```
PRINT A,B,NEWPAGE,C
```

results in only the value of C being displayed. The simple statement PRINT NEWPAGE can be used to clear the screen.

When PRINT without USING refers to array data, the following applies:

1. Each array is started on a new line and is printed with the rightmost subscripts varying most rapidly.
2. Repetitions of the rightmost subscript's range begin at the start of a new line, and are separated from the preceding line by a single blank line.
3. After the final repetition of the rightmost subscript's range has been printed, a blank line is generated, and the terminal is repositioned to a new print line.

Within each line the separation of values is controlled by the delimiter following the array name.

A single dimensional array prints as a column vector.

PRINT with USING IMAGE Clause

When the USING IMAGE clause is present on the PRINT statement, the printed output is under control of these factors:

- The line size dictated by the terminal
- The current settings of the margins
- The use of commas and semicolons in the output list
- The number of data items in the output list
- The image defined by the USING clause

The current setting of the margins can be controlled by the MARGIN statement. See "MARGIN Statement" on page 178.

OUTPUT-LIST—SCALAR ITEMS: Each scalar reference in the output-list is edited into that portion of a line as directed by the IMAGE. The first position is determined by the left margin, and the last position cannot be beyond the right margin.

If the output-list contains at least one item, there must be at least one conversion specification in the referenced IMAGE clause.

If the output list contains no items and the IMAGE contains no characters, a blank line is printed.

If the output list contains no items and the IMAGE contains characters, the IMAGE is printed, up to the first unused conversion specification.

If the number of scalar references in the output list is less than the number of conversion specifications in the IMAGE, the output image is ended at the first unused conversion specification and the remainder of the IMAGE is ignored.

If the number of scalar references in the output list exceeds the number of conversion specifications in the IMAGE, and if the scalar reference using the last specification is followed by:

- A semicolon, the IMAGE is reused from its beginning for the remaining scalar references, in the next position of the current print line.
- A comma, the current line is printed and the IMAGE is reused for the remaining scalar references on the next print line.

If the number of scalar references in the output-list does not exceed the number of conversion specifications in the IMAGE, and the last scalar reference is followed by:

- A semicolon, the next output from a PRINT statement is begun at the next position of the current line.
- A comma, the next output is begun in the next print zone of the current line.

If the last scalar reference is not followed by either a comma or a semicolon, the current line is ended and the next PRINT statement output is on the next line.

OUTPUT-LIST—ARRAY ITEMS: When an array appears in the output-list, the beginning of the array is started on a new line to separate the output from the preceding line. The array elements are printed with the rightmost subscript varying most rapidly. Completion of the range of the rightmost subscript, forces the end of the current line, generation of a blank line, and reuse of the IMAGE on a new line. After the last iteration of the rightmost range is printed, a blank line is written and the position is reset to the beginning of the next line.

If the number of array members in the range of the rightmost subscript exceeds the number of conversion specifications in the IMAGE, the IMAGE is reused. In this case, if the array name in the output list is followed by:

- A semicolon, each reuse of the IMAGE is on the same line.
- A comma, each reuse of the IMAGE is on a new line.

If the number of rightmost range members is less than the number of conversion specifications, the line is terminated at the first unused conversion specification.

Single dimension arrays (vectors) are displayed as a column.

For scalars mixed with arrays in the output list, each consecutive set of scalars causes the set of scalar values to be printed at the start of a line, from the beginning of the IMAGE.

Example

```
110 OPTION BASE 1
120 DIM A(2,2),B(3)
130 MAT A=(1)
140 MAT B=(2)
150 PRINT USING 160:MAT A,3,4,MAT B,5,6,7,8,9
160 :__#__#__#__#
```

The above example produces the following output:

```
__1__1
blank line
__1__1
blank line
__3__4
_2
blank line
_2
blank line
_2
blank line
__5__6__7__8
__9__
```

PRINT with USING FORM Clause

When the USING FORM clause is present on the PRINT statement, the printed output is under control of these factors:

- The line size dictated by the terminal
- The current settings of the margins
- The use of semicolons and commas in the output list
- The number of items in the output list
- The FORM definition

The current setting of the margins can be controlled by the MARGIN statement. See "MARGIN Statement" on page 178.

At the beginning of the construction of the print record, the entire record contains blanks. Individual data fields are superimposed over these blanks, according to the data form and control specifications of the FORM specification. Each item of the output list is matched against the corresponding data form specification, converted if necessary to the form and length indicated, and placed in the position specified by the control specifications.

If the number of items in the output-list is less than or equal to the number of data form specifications in the FORM specification, any control specifications immediately following the last data form specification used are also used.

If the number of items in the output-list exceeds the number of data form specifications, any trailing control specifications are used, and then the FORM is reused from its beginning.

A valid PAGE control specification always causes the output of the current line.

Except for SKIP 0, a valid SKIP control specification causes the output of the current line.

If the output-list has been exhausted, the current line is output unless the list ends with:

- A comma, which positions the line to the next print zone.
- A semicolon, which positions the line to the next print position.

If the output-list is not exhausted (that is, the FORM is to be reused):

- A comma after the last item processed in the output list writes the current line.
- A semicolon after the last item processed in the output list positions the line to the next position.

If the position of a value to be displayed will start beyond the right margin, the current line is written.

If the display of a value is begun but cannot be completed within the right margin, the line with that portion of the value is printed, and the remainder of the value begins on the next line.

In all cases, after an output line is displayed, the line position is reset to the initial position of the next line.

Print-associated FORM statements may specify the control specifications X, POS, SKIP, and PAGE, and the data form specifications C, N, V, and PIC. See "FORM Statement" on page 129.

Immediate Execution

The PRINT statement can be used to display the values of variables and arrays created by the program or by other immediate statements.

All features of the PRINT statement may be used with the following restrictions:

- The USING clause, if any, cannot refer to an IMAGE or FORM statement in the workspace. It must be a character expression.
- The err clauses (EXIT, CONV, IOERR, SOFLOW) are not allowed, because such clauses refer to program line numbers or statement labels in the workspace.

PRINT FIELDS STATEMENT (FOR FULL SCREEN TERMINAL DISPLAY)

The PRINT FIELDS statement displays one or more data values on a display terminal screen in the specified screen field(s).

Format

```
PRINT [#fileref[,]] FIELDS field-definition:
      output-list [;] [err[,err]...]
```

Where:

fileref
is a numeric expression whose rounded integer value evaluates to zero.

field-definition
can be:

character expression

or

MAT character array name

Each character expression or character array name must evaluate to:

"row, column[,[data-form][,[leading][,trailing]]]"

Where:

row
is a positive nonzero integer, specifying the row of the display

column
is a positive nonzero integer, specifying the first column of the display

data-form
can be one of the data forms shown in Figure 40 on page 226.

leading
are display attributes for the print field

trailing
are display attributes for the positions between the print field and the next field.

Display attributes that have meaning to IBM BASIC are:

H
highlighted

I
invisible (not displayed)

N
normal intensity

Note: For ease of migration from other BASIC products, B, R, and U are also accepted and treated as N (normal intensity). Multiple attribute characters may be specified. Unrecognized characters are ignored. If I is specified, it overrides H and N. H overrides N. N is the default.

Data Form	Meaning
W	Length of data item.
C[W]	Character data.
V[W]	Character data.
NW[.d]	Conversion of numeric data to character data.
G[W[.d]]	Represents either character data or conversion of numeric data to character data, depending upon whether the type of the data is character or numeric.
PIC(s[s]...[¬¬¬[¬...][tr]])	Picture of data item

Where:

W
is an unsigned, nonzero integer constant, which may be preceded with blanks.

d
is an unsigned, integer constant.

s
is a digit specifier (#, Z, *, $, +, or -), or an insertion character (a comma (,), solidus (/), blank (B), or decimal point (.).

¬
is an exponent specifier, where three or more (¬) characters are shown. (Can also be specified as the circumflex character.)

tr
is a trailing character, that is, a trailing plus (+), trailing minus (-), trailing credit (CR), or either form of trailing debit (DB or DR).

Note: The total length of w, in characters, can be from 1 through (screen-width - number-of-attributes) for character data, or 1 through 156 for numeric data. If w (or w.d) is omitted, the length is 1 character.

Where:

screen-size
is the total number of characters on the screen

number-of-attributes
is:

0 if neither leading nor trailing attributes are specified.

1 if a leading attribute or a trailing attribute (but not both) is specified.

2 if both leading and trailing attributes are specified.

Figure 40. PRINT FIELDS Statement—Data Form Codes

output-list
is a list of zero or more constants, variables, array elements, expressions (numeric and character), and/or entire arrays (prefaced with MAT). List elements are separated by commas.

err
can be one of the following:

EXIT line-ref

CONV line-ref

IOERR line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

When the PRINT FIELDS statement is executed, the data defined in the output-list is displayed on the terminal screen in the positions specified by the field-definition.

If the terminal does not have a screen, an IOERR exception occurs. (See "Full Screen Input/Output Statements" on page 73.)

FILEREF: The fileref is a numeric expression that should, when rounded to an integer, evaluate to zero; if it does not, an IOERR exception occurs. The standard system action is to replace the value with zero.

FIELD-DEFINITION: A field-definition entry can be a character expression or MAT character array name:

- If a field-definition entry is a character expression, it defines one print field, and only one item can be displayed.
- If a field-definition entry is MAT character array name, it can define one or more print fields. In this case, the field-definition entry must be a one dimensional array; the field-definition entries within the array need not match the order of the fields on the screen.

If an array is specified for a field-definition entry, the number of fields is the number of output-list items, not the number of elements in the array. The number of elements in the array can exceed the number of output-list items; any extra array elements are ignored. However, all of the array elements are syntax checked.

Row and column are positive, nonzero integers that specify the starting location of the field. Row 1, column 1 is the upper left-hand corner of the screen. If row or column is greater than the dimensions of the screen, an exception occurs.

The data-form specifies the number of characters in the screen field, that is, the length of the screen field. A field that extends beyond the rightmost column is continued starting in column 1 of the next row, the bottom row continuing in the top row of the screen.

If the data-form specification is omitted, the length of the field is:

- For character items, the character length of the output-list item
- For numeric items, the length of the output-list item if it was printed by a PRINT statement without the USING clause

For the C, V, and G data forms, the length of the field (w or w.d) may be omitted from the data-form specification. If the length is omitted, the field is one character long.

If the data-form definition includes a length specification, the data is displayed as follows:

- Character strings are left-justified in the field with blanks padded on the right. If the string length is greater than the field length, a string overflow exception occurs.
- Numeric values are displayed according to the rules for the data-form specified as shown in Figure 40 on page 226. If the field size is smaller than the expression value, asterisks fill the field and an exception occurs.

Display Attributes specify how the display is treated.

Leading Display Attributes specify how the print field is to display on the screen. If specified, the leading attribute occupies one character position on the screen, preceding the field.

Trailing Display Attributes specify how the positions between the print field and the next field are to display. If specified, the trailing attribute occupies one character position on the screen, following the field.

The location of the trailing display attribute for one field can overlap with the leading display attribute of the following field. If leading and trailing attributes overlap, the last attribute written to the screen is the one in effect.

The default display attribute is N; all other attributes override it. The I attribute overrides all other display attributes.

A set of leading or trailing attributes should not be separated by commas; the comma specifies the beginning and ending of each leading or trailing list.

The attributes can be entered in any order.

OUTPUT-LIST: The output-list can be omitted; if it is omitted, nothing is displayed.

If the output-list entry is not an array name, only one item of data can be displayed for that entry.

If the output-list entry is an array name, items are taken from the list on a row-by-row basis and displayed on the screen as specified by the field-definition.

OPTIONAL SEMICOLON: The optional semicolon after the output-list indicates that the display field should be saved, but not displayed on the screen until one of the following occurs:

- A PRINT FIELDS without a semicolon is executed
- Any other input/output statement which accesses the screen is executed (PRINT, INPUT, INPUT FIELDS, etc.)
- Some external stimulus (external to BASIC) causes the screen display to change, for example, pressing the CLEAR key.

ERROR CONDITIONS: If a string overflow occurs, the SOFLOW exception occurs. If a numeric conversion cannot be performed as required, the CONV exception occurs. If a hardware malfunction prevents completion of the display, the IOERR exception occurs.

These exceptions can be recovered from, if the CONV, IOERR, or SOFLOW clauses are specified, or if an EXIT clause refers to an EXIT statement that contains these clauses.

The I/O error conditions interact with the ON Condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example 1

```
100 PRINT FIELDS "10,12,C 15" :PASSWORD$
```

Displays the value of PASSWORD$ on the terminal screen, beginning at row 10, column 12. If the data in PASSWORD$ is less than 15 characters in length, the balance of the specified area is space filled.

Example 2

```
100 A$ = "22,1,C 3"
110 PRINT FIELDS A$: "AGE"
```

Prints AGE at location row 22, beginning at column 1.

Example 3

```
130 DATA "10,10,C10,H", "12,20,C10",&
  & "14,20,C10", "16,20,C20,N"
140 MAT READ A$
      .
      .
      .
180 PRINT FIELDS MAT A$: "NAME", "ADDRESS", "CITY",&
  & "STATE, ADDRESS-CODE"
```

This PRINT FIELDS statement prints four fields according to the field-definitions specified in array A$. See Example 3 in "INPUT FIELDS Statement (For Full Screen Terminal Input)" on page 161 for the program in which this statement is used.

PRINT FILE STATEMENT (FOR DISPLAY FORMAT FILES)

The PRINT File statement transmits data to a display format file.

Format

```
[MAT] PRINT #fileref [[,]USING line-ref]
            [[,]FONT expression]: [output-list]
            [err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 0 to 255 and which identifies the file to be processed.

line-ref
is the line number or line label of an IMAGE or FORM statement, or a character expression which contains a FORM or an image.

expression
is a numeric expression with a rounded integer value of 1 to 4.

Note: The fileref, USING, and FONT clauses may occur in any sequence.

output-list
is a list of constants, variables, array elements, expressions (numeric and character), and entire arrays (prefaced with MAT). List elements are separated by commas or semicolons. The keywords TAB (followed by a numeric expression in parentheses), PAGE, and NEWPAGE may be used in the list.

err
is one of the following:

EXIT line-ref

ENDPAGE line-ref

PAGEOFLOW line-ref

EOF line-ref

IOERR line-ref

CONV line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

The colon may be omitted if it would be the last nonblank character on the line.

Example

```
100 PRINT #1:
```

and

```
100 PRINT #1
```

are equivalent, and result in a blank record.

Description

When used as a file statement, the PRINT statement must refer to a file having display format. If DEVICE PRINTER or DEVICE 3800 is specified in the file-id of the OPEN statement, each output record is prefixed with a carriage control character. If DEVICE 3800 is specified, a font control character is prefixed after the carriage control character.

MAT KEYWORD: The MAT keyword preceding the PRINT keyword specifies that the output-list consists only of arrays; the MAT keyword is then unnecessary in the output-list. See "Input/Output Lists" on page 70 for more information.

FILEREF: The fileref must refer to a display format file. (See "Combinations of File Organization and Format" on page 57.)

USING CLAUSE: The IMAGE and FORM statement considerations for a PRINT File statement are exactly the same as for a PRINT statement. See "PRINT Statement" on page 217.

FONT CLAUSE: The FONT clause is used in conjunction with the DEVICE 3800 clause in the fileid of the OPEN statement to specify which font on the 3800 printer is to be used.

OUTPUT-LIST: The output-list is a set of items separated by commas or semicolons. The list may include the TAB and NEWPAGE clauses, if no USING clause is present:

```
TAB(e) where e is a numeric expression
[NEW]PAGE
```

Use of the TAB clause is explained in "PRINT Statement" on page 217.

Use of PAGE or NEWPAGE forces the beginning of a new record, with a carriage control character for page eject. If PAGE or NEWPAGE is specified and the file is not opened as DEVICE PRINTER or DEVICE 3800 (see "OPEN Statement" on page 206), an exception is generated. The SYSTEM action for this exception is a warning message.

ERROR CONDITIONS: The error conditions IOERR, CONV, and SOFLOW, as well as the EXIT reference, function exactly as they do for the PRINT statement. See "PRINT Statement" on page 217.

The EOF condition, which occurs if there is not enough room on the file for a record, may be recoverable if it is included as an err condition.

The ENDPAGE (or PAGEOFLOW) condition occurs if the PRINT File attempts to start a new line beyond the bottom margin. (See "MARGIN Statement" on page 178.)

A PRINT File statement with fileref 0 is equivalent to a PRINT to the terminal. The EOF and ENDPAGE conditions have no meaning in this case. If specified, they are ignored.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

PUT File Statement

PUT FILE STATEMENT

The PUT File statement places values in a stream file.

Format

```
[MAT] PUT #fileref : output-list
          [err[,err]]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 1 to 255, and which identifies the file to be processed.

output-list
is an output list of items separated by commas.

err
is one of the following:

EXIT line-ref

EOF line-ref

IOERR line-ref

line-ref
is a line number or line label.

The EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and the other err clauses are mutually exclusive.

Description

The PUT statement writes the values specified by the items in the output list into a stream file. Both character and numeric values are stored in internal format and are preceded by an identifying byte. Values from arrays are written with the rightmost subscripts varying most rapidly.

MAT KEYWORD: The MAT keyword preceding the PUT keyword specifies that the output-list consists only of arrays; the MAT keyword is then unnecessary in the output-list. See "Input/Output Lists" on page 70 for more information.

FILEREF: The fileref must refer to a stream file. (See "Combinations of File Organization and Format" on page 57.)

ERROR CONDITIONS: If values remain to be written on the file but, space for the file is exhausted, an EOF (end-of-file) condition exists.

If a hardware malfunction or other condition prevents the PUT statement from completing execution, an IOERR condition exists. Examples of IOERR are: attempting to execute a PUT statement on a file opened for INPUT, or on a fileref 0.

Both EOF and IOERR conditions may be recoverable if the corresponding err clause is used in the PUT statement or in a referenced EXIT statement.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 PUT #12: A_NUMBER#,A_STRING$ EXIT 200
200 EXIT IOERR 900,EOF 1000
```

The above PUT statement adds a numeric and a string value to the stream file associated with file reference number 12. IOERR conditions are handled at line number 900, and EOF conditions at line number 1000.

RANDOMIZE Statement

RANDOMIZE STATEMENT

The RANDOMIZE statement generates a new starting point for the list of pseudorandom numbers used by the RND function.

Format

```
RANDOMIZE
```

Description

RANDOMIZE establishes a new seed value for the intrinsic RND function, much the same as the RND(x) version of the function sets a new seed. The difference is that the RANDOMIZE seed is random, that is, unpredictable.

Immediate Execution

The immediate RANDOMIZE statement operates with the same restrictions and capabilities as the nonimmediate RANDOMIZE statement, as described above.

READ STATEMENT

The READ statement retrieves the internal data files created by DATA statements.

Format

```
[MAT] READ input-list [err[,err]]
```

Where:

input-list
is an input list of items separated by commas.

err
is one of the following:

EXIT line-ref

CONV line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

When a READ statement is executed, successive values are assigned from the internal data file to the items in the input list. If all of the values of the data file have been used and unassigned items remain in the input list of a READ, an exception occurs; however, a RESTORE statement can be used to reset the pointer to the beginning of the data file.

See also "DATA Statement" on page 105.

MAT KEYWORD: The MAT keyword preceding the READ keyword specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input-list.

See "Input/Output Lists" on page 70 for more information.

INPUT-LIST: The input-list can consist of character or numeric variables or arrays.

The length of a character variable is set to the length of the character data assigned to it; a string overflow occurs if the length of the data exceeds the maximum length of the character variable.

Numeric variables must be assigned numeric values. If a numeric value exceeds the defined maximum of its corresponding data type, numeric overflow occurs (OFLOW). If the numeric value is less than the defined minimum, then numeric underflow occurs (UFLOW).

References to entire arrays (MAT) in the input list are assigned from the data file with the rightmost subscript varying most rapidly, starting at the current data file position. If optional expressions follow the array names, the rounded integer portion of the expressions is used to redimension the arrays before the values are assigned.

ERROR CONDITIONS: Conversion and string overflow errors may be recovered from if the appropriate err clause is included in the READ statement or a referenced EXIT statement. Numeric overflow and underflow can be handled by the ON Condition statement.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
  90 OPTION BASE 1
 100 DATA ABCD,123.4,2,4,"XYZ",7*"ZZZ"
 200 READ A$,B#
 300 DIM Z$(10,10)*3
 400 READ P%,Q%,MAT Z$(P%,Q%)
```

In the above example, the first READ assigns the value "ABCD" to A$ and 123.4 to B#. The second READ first assigns the value 2 to P% and 4 to Q%; the array Z$ is redimensioned to eight members and assigned the remaining values XYZ, and 7 repetitions of ZZZ, in the data file.

READ FILE STATEMENT

The READ File statement retrieves records from native and internal files.

Format 1 (native files)

```
[MAT] READ #fileref [,]USING line-ref
           [[,]pos]:
           input-list [err [,err]...]
```

Format 2 (internal files)

```
[MAT] READ #fileref: input-list [,SKIP REST]
           [err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 1 to 255 and which identifies the file to be processed.

line-ref
is the line number or line label of a FORM statement, or a character expression containing a FORM.

pos
is one of the following:

KEY [rel] character expression

SEARCH [rel] character expression

REC[ORD] [=|EQ]numeric expression

rel
is =, =>, >=, EQ, or GE.

If rel is omitted, = is assumed.

Note: The fileref, USING, and pos clauses can occur in any sequence.

input-list
is an input list of items separated by commas.

err
is one of the following:

EXIT line-ref

EOF line-ref

IOERR line-ref

CONV line-ref

SOFLOW line-ref

NOREC line-ref

NOKEY line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

The execution of a READ statement involves two steps

1. Retrieving a record and placing it in a buffer
2. Assigning values from the buffer to the items in the input list

For both native files and internal files, the next sequential record is retrieved by the READ statement with no KEY, SEARCH, or RECORD clause; for relative files, the next sequential record is the next non-null record, and for keyed files, it is the next record in key-sequence.

For a relative file, a specific record can be retrieved through the RECORD clause.

For keyed files, a KEY or SEARCH clause can be used, and the record retrieved is the first record which satisfies the condition specified in the clause:

- With the KEY clause, the key length in the record and the length of the string specified in the condition must be the same.
- With the SEARCH clause, the length of the specified string can be less than the key length, and only that number of high-order positions are compared.

Once the record has been placed in the buffer, values are assigned from the buffer to the list of variables:

- For internal files, this assignment is done in the same manner as the LET statement assigns values to variables.
- For native files, the values are formatted according to the specifications of a FORM statement; this allows the conversion of data in a variety of external representations to internal representations.

For both types of files, each value read and assigned must be of the same basic type (character or numeric) as the corresponding variable in the input list; although a numeric value may be read into a character variable.

MAT KEYWORD: The MAT keyword preceding the READ keyword specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input-list.

See "Input/Output Lists" on page 70 for more information.

INPUT-LIST: An array in the input list is recognized by the keyword MAT appearing before the array name. If redimension specifications appear after the array name in the input list, the array is first redimensioned to extents equal to the rounded integer values of the numeric redimension expressions, and then the array is filled. When an array is redimensioned, the original number of members may not be exceeded.

ERROR CONDITIONS: Various error conditions can occur as values from the record buffer are assigned to list items.

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

The execution of a READ statement involves two steps

1. Retrieving a record and placing it in a buffer
2. Assigning values from the buffer to the items in the input list

For both native files and internal files, the next sequential record is retrieved by the READ statement with no KEY, SEARCH, or RECORD clause; for relative files, the next sequential record is the next non-null record, and for keyed files, it is the next record in key-sequence.

For a relative file, a specific record can be retrieved through the RECORD clause.

For keyed files, a KEY or SEARCH clause can be used, and the record retrieved is the first record which satisfies the condition specified in the clause:

- With the KEY clause, the key length in the record and the length of the string specified in the condition must be the same.
- With the SEARCH clause, the length of the specified string can be less than the key length, and only that number of high-order positions are compared.

Once the record has been placed in the buffer, values are assigned from the buffer to the list of variables:

- For internal files, this assignment is done in the same manner as the LET statement assigns values to variables.
- For native files, the values are formatted according to the specifications of a FORM statement; this allows the conversion of data in a variety of external representations to internal representations.

For both types of files, each value read and assigned must be of the same basic type (character or numeric) as the corresponding variable in the input list; although a numeric value may be read into a character variable.

MAT KEYWORD: The MAT keyword preceding the READ keyword specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input-list.

See "Input/Output Lists" on page 70 for more information.

INPUT-LIST: An array in the input list is recognized by the keyword MAT appearing before the array name. If redimension specifications appear after the array name in the input list, the array is first redimensioned to extents equal to the rounded integer values of the numeric redimension expressions, and then the array is filled. When an array is redimensioned, the original number of members may not be exceeded.

ERROR CONDITIONS: Various error conditions can occur as values from the record buffer are assigned to list items.

For character data, the length of the receiving variable is set to the length of the string sent to it. However, if the string being sent is longer than the maximum length of the receiving variable, a string overflow occurs. For a native file, this can happen with a Cw FORM specification where w exceeds the maximum length of the receiving variable.

For internal files, if the OPEN statement did not specify stream organization and the items of the input list are all used and more data remains in the record, a conversion exception occurs. However, if a SKIP REST clause is specified, the rest of the data in the record is ignored and the CONV exception is avoided.

For internal files, a conversion exception occurs if there is not enough data in the record to fill all the input items, and if the OPEN statement did not specify stream organization.

For native files, a conversion exception occurs in both these situations.

Other CONV exceptions occur if a value cannot be converted to the type of variable specified, or if an attempt is made to reference a location outside a native file record with a POS or X control specification.

An EOF (end-of-file) exception occurs when no KEY, SEARCH, or REC is specified and the last record of the file has already been read.

The NOKEY and NOREC exceptions occur when no record exists which satisfies the KEY or SEARCH condition on a keyed file, or the RECORD condition on a relative file.

The IOERR exception occurs if a hardware malfunction or other error which prevents the reading of the record.

If the SKIP REST clause is specified and the input record contains more data than necessary to satisfy all of the items in the input list, the remainder of the record is ignored. If SKIP REST is not specified and this condition occurs, a CONV exception is generated.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 READ #TWO%: A$, B%, SKIP REST
200 READ #3 USING 300: C$, D% IOERR 500
300 FORM C20,N5,X 30
400 DIM ARR#(10,10)
500 READ #4 USING 600 KEY="XYZ":P%,Q%,MAT ARR#(P%,Q%-P%)
600 FORM N2,N2,6*N5
```

In the above example, the first READ assigns values from the next sequential record of an internal file to two variables, and ignores the rest of the record. The second read performs the same function for a record of a native file, and will pass control to line number 500 if an IOERR condition occurs. The third read retrieves a record with key "XYZ" from a keyed file; the first two values are used to redimension the array ARR#, and then the array is filled with numeric values.

REM Statement

REM STATEMENT

The REM statement inserts remarks into a program.

Format

```
REM [remark]
```

Where:

remark
is any character string.

Description

A REM statement may be placed anywhere within a program. It is a nonexecutable statement and its line number or line label may be used as the target for transfer of control statements, for example, GOTO. Execution then continues with the next executable statement after the REM statement.

Since any character is permitted within a remark, each remark is considered terminated at end-of-line. This means:

- REM is the last statement on a line (a following statement separating colon is not recognized).
- REM statements cannot be continued (the continuation ampersand is not recognized).

Comments Using the Exclamation Mark

Remarks may also be appended to statement lines by using an exclamation mark as a statement delimiter. A statement beginning with an exclamation mark is treated the same as a REM statement.

The remark clause (!) is nonexecutable. It appears in the statement to indicate that the data following is to be considered a remark only. It has no effect on program execution. It may not appear on either an IMAGE, DATA, or FORM statement since the exclamation mark can be meaningful within those statements.

Example

```
100 IF NUMBER LT 200 THEN GO TO 200
110 REM TEST NUMBER

is functionally equivalent to

100 IF NUMBER LT 200 THEN GOTO 200,   !TEST NUMBER
```

As with the REM statement, a trailing comment must be the last entry on a line. Unlike REM statements, a line with a trailing comment can be continued; an ampersand as the last nonblank character on the line indicates continuation. However, it is not the trailing comment itself that is continued; it is the statement before the trailing comment that is continued.

Example

```
100 OPEN #4: NAME "PARTS",!FILE OF PART DESCRIPTIONS &
  & ORGANIZATION RELATIVE, !RECORD NUMBERS ARE &
  & TYPE NATIVE, ACCESS OUTIN !PART NUMBERS.
```

is functionally equivalent to:

```
100 OPEN #4: NAME "PARTS",ORGANIZATION RELATIVE, &
& TYPE NATIVE, ACCESS OUTIN
```

REREAD Statement

REREAD STATEMENT

The REREAD File statement makes the last accessed record in a native file available again.

Format

```
[MAT] REREAD #fileref [,]USING line-ref:
                input-list [err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, is a positive integer with the range 1 to 255, and which identifies the file to be processed.

line-ref
is the line number or line label of a FORM statement, or a character expression containing a FORM.

Note: The fileref and USING clauses may occur in any order.

input-list
is an input list of items separated by commas.

err
is one of the following:

EXIT line-ref

IOERR line-ref

CONV line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

The EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

The REREAD statement can only be used for native files. The last access to the specified file must have been either a READ statement or another REREAD statement.

MAT KEYWORD: The MAT keyword preceding the REREAD keyword specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input-list.

See "Input/Output Lists" on page 70 for more information.

FILEREF: The fileref must refer to a native file. (See "Combinations of File Organization and Format" on page 57.)

USING CLAUSE: The data in the record is formatted according to the specifications of the FORM statement.

INPUT-LIST: The data in the record is assigned to the variables in the input-list, in the same manner as the READ statement for files. (See "READ FILE Statement" on page 237.)

REREAD STATEMENT

The REREAD File statement makes the last accessed record in a native file available again.

```
Format

[MAT] REREAD #fileref [,]USING line-ref:
                input-list [err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, is a positive integer with the range 1 to 255, and which identifies the file to be processed.

line-ref
is the line number or line label of a FORM statement, or a character expression containing a FORM.

Note: The fileref and USING clauses may occur in any order.

input-list
is an input list of items separated by commas.

err
is one of the following:

EXIT line-ref

IOERR line-ref

CONV line-ref

SOFLOW line-ref

line-ref
is a line number or line label.

The EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

The REREAD statement can only be used for native files. The last access to the specified file must have been either a READ statement or another REREAD statement.

MAT KEYWORD: The MAT keyword preceding the REREAD keyword specifies that the input-list consists only of arrays; the MAT keyword is then unnecessary in the input-list.

See "Input/Output Lists" on page 70 for more information.

FILEREF: The fileref must refer to a native file. (See "Combinations of File Organization and Format" on page 57.)

USING CLAUSE: The data in the record is formatted according to the specifications of the FORM statement.

INPUT-LIST: The data in the record is assigned to the variables in the input-list, in the same manner as the READ statement for files. (See "READ FILE Statement" on page 237.)

ERROR CONDITIONS: The err conditions IOERR, CONV, and SOFLOW may be recoverable if they are specified on the statement or in a referenced EXIT statement.

A CONV error occurs when a field cannot be converted as specified, there is not enough data in the record, or there is an attempt to reference a location outside the record.

SOFLOW occurs with a string overflow.

IOERR occurs when a hardware malfunction or other error prevents rereading the record.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 READ #2, USING 300: A$, C%
200 REREAD #2 USING 0400: B$, D% EXIT 500
300 FORM C10,POS 21, N5, X 5
400 FORM X 10, C10, POS 26, N5
500 EXIT IOERR 900, CONV 900, SOFLOW 900
```

In the above example, a record in a native sequential file contains four values, two of which are accessed by the READ statement and two by the REREAD statement. If any errors occur on the REREAD statement, control passes to line number 900.

RESET STATEMENT

The RESET statement changes the position of the file pointer.

Format

```
RESET #fileref [[,]pos] :[err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 0 to 255, and which identifies the file to be processed.

pos
is one of the following:

BEGIN

END

APPEND

KEY [rel] character expression

SEARCH [rel] character expression

REC[ORD] [{= |EQ}] numeric expression

rel
is =, >=, =>, EQ, or GE.

If rel is omitted, = is assumed.

Note: The fileref and pos clauses may be in any sequence.

err
is one of the following:

EXIT line-ref

IOERR line-ref

NOREC line-ref

NOKEY line-ref

line-ref
is a line number or line label.

EXIT and all other err clauses are mutually exclusive.

An EXIT clause must refer to the line number or line label of an EXIT statement.

The keyword RESTORE may be used instead of RESET.

The colon may be omitted if it would be the last nonblank character on the line.

Example

```
200 RESET #3, REC = COUNT%:
    and
200 RESET #3, REC = COUNT%
```

are equivalent.

Description

Any type of external file may be repositioned with a RESET statement.

An attempt to RESET fileref 0 is ignored.

POS OPTION: The BEGIN clause positions any file to its beginning; if no positioning clause is specified, BEGIN is assumed.

The END clause positions files, other than relative and keyed, to their end, so that new records can be added; the APPEND clause is identical to END.

For relative files, the RECORD clause positions the file to the record whose relative number is specified.

For keyed files, either the KEY clause or the SEARCH clause positions the file to the first record whose key satisfies the specified condition:

- The KEY clause condition specifies the entire key field,
- The SEARCH clause condition specifies that part of the key with a string length equal to that of the search argument.

ERROR CONDITIONS: The error conditions IOERR, NOREC, and NOKEY may be recovered if they are included in err clauses or a referenced EXIT statement.

The NOREC condition occurs if no relative record satisfies the RECORD condition for a relative file.

The NOKEY condition occurs if no keyed record satisfies the KEY or SEARCH condition for a keyed file.

The IOERR condition occurs if the file cannot be repositioned for some other reason.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 RESET #10, KEY="1234": NOKEY 900
     .
     .
     .
500 RESET #10: IOERR 1000
```

At statement 100, a keyed file is positioned to the record whose key is "1234". If no such record exists, control is transferred to line number 900. At statement 500, the same file is repositioned to its beginning; in the event of an IOERR, control is transferred to line number 1000.

RESTORE STATEMENT

The RESTORE statement resets the file pointer of an internal data file to its first value.

Format

```
RESTORE
```

Description

The RESTORE statement resets the file pointer of an internal data file, created by DATA statements and accessed by READ statements.

The RESTORE statement resets the internal data pointer to the first value in the internal file.

The RESTORE statement is ignored if there are no DATA statements in the program unit.

(See also "RESET Statement" on page 244 for other uses of the RESTORE keyword.)

Example

```
100 DATA 123,456,789
200 READ A%, B%
300 RESTORE
400 READ C%
```

In the above example, the value assigned to C% is 123, because at line number 300 the data file was repositioned to its first value.

RETRY STATEMENT

The RETRY statement reprocesses statement which caused an exception.

Format

```
RETRY
```

Description

Execution of a RETRY statement results in reprocessing the statement which caused an exception. The RETRY statement provides for a return to normal requested statement execution after program flow has been diverted to process an exception.

If an exception condition does not exist when the RETRY statement is executed, an exception occurs.

Example

```
100 ON ZDIV GOTO 1000
 .
 .
 .
500 BAL = A - B
510 DIVI = TOT/BAL
520 BAL = A + B
 .
 .
 .
1000 BAL = 1
1010 RETRY
```

Statement 100 sets the condition being tested. If BAL is set to zero at statement 500, execution of 510 will trigger the ZDIV (divide by zero) condition. Execution will branch to statement 1000, set BAL to 1, and return to statement 510 because of the RETRY statement.

RETURN STATEMENT

The RETURN statement returns control to the next executable statement following the GOSUB statement that called the subroutine.

Format

```
RETURN
```

Description

When a RETURN statement is executed, program execution is returned to the next statement following the last GOSUB statement executed, completing a GOSUB/RETURN cycle. (See "GOSUB Statement" on page 145.)

Execution of a RETURN statement returns the last active GOSUB statement to inactive status.

The execution of a RETURN statement without an active GOSUB statement results in an exception.

It is not necessary that equal numbers of GOSUB statements and RETURN statements be executed before termination of a program unit or multiline function (execution of a SUBEXIT statement in a subprogram or of a FNEND statement in a function). All active GOSUB statements associated with a program unit or function are set inactive upon termination of the program unit or function.

For more information, see "Subroutine Control Statements" on page 61.

REWRITE STATEMENT

A REWRITE statement updates a record stored in a native file.

```
Format

    [MAT] REWRITE #fileref [,]
                  USING line-ref [[,]pos]:
                  output-list [err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 1 to 255, and which identifies the file to processed.

line-ref
is the line number or line label of a FORM statement, or a character expression which contains a FORM.

pos
is KEY [=|EQ] character expression

or

REC[ORD] [=|EQ] numeric expression

Note: The fileref, USING, and pos clauses may occur in any sequence.

output-list
is an output list of variable or array names of items to be output, separated by commas.

err
is one of the following:

EXIT line-ref

IOERR line-ref

EOF line-ref

CONV line-ref

SOFLOW line-ref

NOREC line-ref

NOKEY line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

The REWRITE statement updates records of native files which have been opened with the access attribute OUTIN.

MAT KEYWORD: The MAT keyword preceding the REWRITE keyword specifies that the output-list consists only of arrays; the MAT keyword is then unnecessary in the output-list.

See "Input/Output Lists" on page 70 for more information.

FILEREF: The fileref must refer to a native file opened with access OUTIN. (See "Combinations of File Organization and Format" on page 57.)

USING CLAUSE: The USING clause identifies the line number or line label of the FORM statement to be used in formatting the record.

The USING clause can itself contain a character expression that specifies a format for the data.

POS CLAUSE: If the RECORD clause is included for a relative file, or if the KEY clause is included for a keyed file, the specified record is read, updated, and rewritten. If the record belongs to a keyed file, an IOERR condition occurs if the value of the key is changed.

If no RECORD or KEY clause is specified, the last access to the file must have been a READ or REREAD of the record to be written.

OUTPUT-LIST: The values from the output-list are formatted according to the specifications of the FORM statement and replace whatever data previously occupied those positions of the record. Portions of the original record can be preserved by using the X and POS control specifications of the FORM statement.

For relative and sequential files, the length of the new record resulting from the output-list, or from the interaction of the output-list and the FORM, must not exceed the length of the original record. For keyed files, however, the new record length may be greater than the original length.

ERROR CONDITIONS: Several error conditions may be recovered from if an err clause for the condition is specified in the statement or the referenced EXIT statement.

A NOKEY or NOREC condition occurs if the key or record number specifies a record which does not exist.

A CONV condition occurs if a value cannot be converted as specified in the FORM.

The SOFLOW condition is generated by a string overflow.

The EOF condition occurs if the record to be rewritten will no longer fit on the file.

The IOERR condition indicates that a hardware malfunction or other cause prevents the writing of the altered record.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 REWRITE #25 KEY EQ KEY1$ USING 120: KEY1$,A% &
  &     EXIT 140
120 FORM C4,X 10,N6,POS 25,"WXYZ"
140 EXIT NOREC 900
```

In the above example of a REWRITE statement for a keyed file, the key is in positions 1-4; positions 5-14 and 21-24 of the original record are unchanged; the value of A% replaces the contents of record positions 15-20, and the value "WXYZ" replaces the contents of record positions 25-28. If no record exists with a key value as specified in KEY1$, then control passes to line 900.

MAT KEYWORD: The MAT keyword preceding the REWRITE keyword specifies that the output-list consists only of arrays; the MAT keyword is then unnecessary in the output-list.

See "Input/Output Lists" on page 70 for more information.

FILEREF: The fileref must refer to a native file opened with access OUTIN. (See "Combinations of File Organization and Format" on page 57.)

USING CLAUSE: The USING clause identifies the line number or line label of the FORM statement to be used in formatting the record.

The USING clause can itself contain a character expression that specifies a format for the data.

POS CLAUSE: If the RECORD clause is included for a relative file, or if the KEY clause is included for a keyed file, the specified record is read, updated, and rewritten. If the record belongs to a keyed file, an IOERR condition occurs if the value of the key is changed.

If no RECORD or KEY clause is specified, the last access to the file must have been a READ or REREAD of the record to be written.

OUTPUT-LIST: The values from the output-list are formatted according to the specifications of the FORM statement and replace whatever data previously occupied those positions of the record. Portions of the original record can be preserved by using the X and POS control specifications of the FORM statement.

For relative and sequential files, the length of the new record resulting from the output-list, or from the interaction of the output-list and the FORM, must not exceed the length of the original record. For keyed files, however, the new record length may be greater than the original length.

ERROR CONDITIONS: Several error conditions may be recovered from if an err clause for the condition is specified in the statement or the referenced EXIT statement.

A NOKEY or NOREC condition occurs if the key or record number specifies a record which does not exist.

A CONV condition occurs if a value cannot be converted as specified in the FORM.

The SOFLOW condition is generated by a string overflow.

The EOF condition occurs if the record to be rewritten will no longer fit on the file.

The IOERR condition indicates that a hardware malfunction or other cause prevents the writing of the altered record.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 REWRITE #25 KEY EQ KEY1$ USING 120: KEY1$,A% &
  &     EXIT 140
120 FORM C4,X 10,N6,POS 25,"WXYZ"
140 EXIT NOREC 900
```

In the above example of a REWRITE statement for a keyed file, the key is in positions 1-4; positions 5-14 and 21-24 of the original record are unchanged; the value of A% replaces the contents of record positions 15-20, and the value "WXYZ" replaces the contents of record positions 25-28. If no record exists with a key value as specified in KEY1$, then control passes to line 900.

SCRATCH STATEMENT

The SCRATCH statement erases the contents of a file.

Format

```
SCRATCH #fileref: [err]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 0 to 255, and which identifies the file to be processed.

err
can be:

EXIT line-ref

or

IOERR line-ref

line-ref
is a line number or line label.

EXIT and IOERR are mutually exclusive.

An EXIT clause must refer to the line number or line label of an EXIT statement.

The colon (after fileref) may be omitted if it would be the last nonblank character on the line.

Example

```
150 SCRATCH #1
```

and

```
150 SCRATCH #1:
```

are equivalent.

Description

Processing the SCRATCH statement erases all the values or records on a file and resets the file pointer to the beginning of the file. An attempt to scratch fileref 0 is ignored.

An IOERR condition occurs if a hardware malfunction or other condition prevents the file from being scratched. If an IOERR err condition is specified in the SCRATCH statement or on a referenced EXIT statement, the IOERR condition may be recoverable.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

SELECT statement

SELECT STATEMENT

The SELECT statement begins a SELECT block.

Format

```
SELECT expression
```

Where:

expression
is a numeric or character expression

Description

The SELECT statement is used with the END SELECT, CASE, and CASE ELSE statements to construct SELECT blocks. SELECT blocks allow for the conditional execution of any one of a number of alternative CASE blocks.

A SELECT block is subdivided into CASE blocks, and each time a SELECT block is entered, one of the CASE blocks or the CASE ELSE block is executed. The value of the expression in the SELECT statement is matched against constants in the CASE statements to determine which CASE block, if any, is selected for execution. This is described under "SELECT Blocks" on page 66.

The SELECT block consists of four parts, coded in the following order:

1. The SELECT line containing the selection expression.
2. Any number of CASE blocks. A CASE block is defined as the statements between one CASE line and another, between a CASE line and a CASE ELSE line, or between a CASE line and an END SELECT line.
3. An optional CASE ELSE block, which includes all statements between the CASE ELSE line and the END SELECT line.
4. An END SELECT line which terminates the SELECT block.

The expression in the SELECT statement is evaluated and its value compared to the CASE items in the CASE statements in the order in which they occur until a match is found. If a match is not found, the CASE ELSE block is executed.

If CASE ELSE is not specified, and no CASE item is matched, an exception occurs.

An example of a SELECT block is given in "SELECT Blocks" on page 66.

STOP STATEMENT

The STOP statement terminates program execution.

Format

STOP [numeric expression]

Where:

numeric expression
is any numeric expression

Description

STOP causes exactly the same action as an END statement during execution. However, unlike the END statement which must be the last physical as well as last logical statement in the main program, the STOP statement may appear anywhere in the program, including a subprogram.

When a STOP statement is executed, all open files are closed.

The numeric expression represents a return code that is returned to the batch operating system. The value returned is the rounded integer evaluation of the expression.

In interactive mode, the value of the numeric expression is displayed as part of the ending message.

Immediate Execution

The immediate STOP statement operates with the same restrictions and capabilities as the STOP statement in a program.

Program files are closed and execution terminated.

The optional numeric expression in the STOP statement is allowed (with the restriction it does not refer to function definitions) but is ignored.

SUB Statement

SUB STATEMENT

The SUB statement is the first statement of a subprogram and names it.

Format

```
SUB name [(parameter[,parameter]...)]
```

Where:

name
is a subprogram identifier of 1 to 7 characters.

parameter
may be either a simple variable or an empty array declarator.

An empty array declarator has the form:

```
identifier ([,]...)
```

Description

The SUB statement is the first line of a subprogram, naming the subprogram and declaring any parameters. The name must not be the same as that used for one of the special CALL formats. (See "CALL Statement" on page 90.)

The number and type of arguments in a CALL statement must agree with the number and type of parameters in the corresponding SUB statement. An array used as an argument must have the same number of dimensions as the corresponding parameter (indicated by the number of commas in an empty array declarator).

An array that is a parameter (that is, appears in a SUB statement) may be redimensioned within a subprogram. When control returns to the calling program, the array retains its changed dimensions.

A given parameter may appear only once in a SUB statement. The parameters cannot appear in DIM or COM statements. The number of dimensions for an array which is a parameter is declared in the SUB statement and the values of the dimensions are those of the corresponding argument at run time. If an argument is an array element, its subscripts are evaluated once, when the subprogram is first invoked. The conventions for passing parameters are given in "CALL Statement" on page 90.

Example

```
100 SUB RTN(A(,))
```

In this example, the SUB statement names the subroutine as RTN, and also declares array A to have two dimensions; however, the values of the dimensions are supplied by the argument at run time.

All parameters and non-COMMON variable and array names specified in a subprogram are local to that subprogram. They are distinct from objects with same names outside the subprogram.

See also "Calling IBM BASIC Programs" on page 80.

SUBEXIT STATEMENT

The SUBEXIT statement stops execution of a subprogram and returns control to the subprogram's caller.

Format

```
SUBEXIT
```

Description

The SUBEXIT statement can only occur within a subprogram. It passes control back to the calling program at the first line following the CALL statement.

See "SUB Statement" on page 254, "CALL Statement" on page 90, and "Subprogram Statements" on page 78.

TRACE STATEMENT

When debugging is active (that is, a DEBUG ON statement has been executed), the TRACE statement turns tracing ON or OFF. The execution of a TRACE statement when debugging is inactive has no effect.

Format

```
TRACE ON [TO #fileref]
    or
TRACE OFF
```

Where:

fileref
is a numeric expression in the range of 0 to 255, indicating the file for the trace listing.

Description

The TRACE statement displays the flow of control within a program.

When TRACE ON and DEBUG ON are in effect, the following actions occur each time a statement of the specified type is executed:

1. For a statement causing a transfer of control both the line number of the statement and the line number of the next statement to be executed (if such a line number exists) are reported.
2. For a statement which changes the value of any variables or arrays, both the line number of the statement and the values assigned to any variables by the statement are reported.

When a TRACE ON statement with a file reference is in effect, trace reports are directed to the file assigned to the specified fileref. If no fileref has been specified, the trace reports are directed to the device associated with fileref zero (the terminal).

If debugging was activated by a DEBUG ON TO fileref statement, the TO fileref clause on a TRACE ON statement is ignored.

A DEBUG OFF statement causes an implicit TRACE OFF, that is, a subsequent DEBUG ON will not resume tracing until another TRACE ON is encountered.

A TRACE OFF statement terminates any file connection set by a TRACE ON TO fileref statement, that is, a subsequent TRACE ON will cause trace output to go to fileref 0.

See also "Debugging Statements" on page 86.

Immediate Trace Execution

The TRACE statement may be executed as an immediate statement. All forms are accepted in the immediate mode. However, if the program unit did not contain both a TRACE ON and a DEBUG ON statement prior to the start of execution, the trace facility will monitor program flow only; it will not show variable assignments. (That is, only the first action described above is done.)

USE STATEMENT

The USE statement establishes a name correspondence in a chained program for those names passed via a CHAIN statement.

Format

```
USE argument-list
```

Where:

argument-list
is a list of scalar and/or array names separated by commas.

Description

Prior to execution of the chained program, the attributes passed for each variable name in the USE statement are matched against the defined attributes for the same variable names in the chaining program. The local variable names in the chained program are then initialized to the passed values.

This matching is done by name, not by order. If a name appears in only one list, it is ignored. If the names match but the types (or number and size of dimensions of an array) do not, an exception occurs. (Passed attributes are not automatically inherited by the chained program. Each variable name in the USE statement must be defined (via DIM or default) within the chained program.)

Example

This statement appears in the chaining program:

```
100 CHAIN "PROGB",VALUEA,VALUEC
```

This statement appears in the chained program (PROGB):

```
200 USE VALUEB,VALUEA
```

Statement 100 ends the chaining program. The data from VALUEA is passed to PROGB, the chained program and placed in the variable named in the USE statement, VALUEA. VALUEB is initialized to the default for its type, because it was not named in the CHAIN statement. PROGB is now executed, with the value passed to it from the chaining program.

Note: The value from VALUEC is not passed to PROGB, although it is included in the CHAIN statement.

The USE statement may appear anywhere in a main program; only one may be specified. The names of COMMON variables may not be included in a USE statement, as COMMON is passed across CHAIN boundaries independently.

The data-list in the USE statement defines the data items retained by the program. All other data values (except COMMON variables) are set to zero or null prior to their first use.

See "CHAIN Statement" on page 98.

WRITE STATEMENT

The WRITE statement adds records or values to native, internal, and stream files.

```
Format 1 (native format)

    [MAT] WRITE #fileref [[,]USING
                line-ref [[,]pos]:
                output-list [err[,err]...]

Format 2 (internal format)

    [MAT] WRITE fileref : output-list
              [err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 1 to 255, and which identifies the file to be processed.

line-ref
is the line number or line label of a FORM statement, or a character expression which contains a FORM.

pos
is REC[ORD] [= or EQ] numeric expression

Note: The fileref, USING, and pos clauses may appear in any order.

output-list
is an output list with items separated by commas.

err
is one of the following:

EXIT line-ref

EOF line-ref

IOERR line-ref

CONV line-ref

SOFLOW line-ref

DUPREC line-ref

DUPKEY line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

WRITE STATEMENT

The WRITE statement adds records or values to native, internal, and stream files.

```
Format 1 (native format)

    [MAT] WRITE #fileref [[,]USING
                line-ref [[,]pos]:
                output-list [err[,err]...]


Format 2 (internal format)

    [MAT] WRITE fileref : output-list
              [err[,err]...]
```

Where:

fileref
is a numeric expression which, when evaluated and rounded, must be a positive integer within the range 1 to 255, and which identifies the file to be processed.

line-ref
is the line number or line label of a FORM statement, or a character expression which contains a FORM.

pos
is REC[ORD] [= or EQ] numeric expression

Note: The fileref, USING, and pos clauses may appear in any order.

output-list
is an output list with items separated by commas.

err
is one of the following:

EXIT line-ref

EOF line-ref

IOERR line-ref

CONV line-ref

SOFLOW line-ref

DUPREC line-ref

DUPKEY line-ref

line-ref
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

EXIT and all other err clauses are mutually exclusive.

Description

The WRITE statement adds records to any sequential file, native, internal, or stream. If the file is positioned at its beginning by an OPEN statement, the WRITE statement places the record at the beginning of the file, replacing any already existing records; in this case, none of the old records are any longer available.

MAT KEYWORD: The MAT keyword preceding the WRITE keyword specifies that the output-list consists only of arrays; the MAT keyword is then unnecessary in the output-list.

See "Input/Output Lists" on page 70 for more information.

FILEREF: The fileref must refer to a native, internal, or stream file, opened for OUTPUT (or OUTIN) access. (See "Combinations of File Organization and Format" on page 57.)

USING CLAUSE: The USING clause is required for writing native files. The record is built by formatting the values from the output-list according to the FORM specifications. If the FORM control specifications X and POS are used, any positions skipped in the record are filled with blanks.

For internal and stream files, the USING clause is invalid; the values are written to the file in internal format.

POS CLAUSE: A WRITE statement with a RECORD clause is used to add a record to a relative file.

For a keyed file, a record is added by a WRITE statement with an output-list containing a character value with the length and position attributes of the key for the file.

ERROR CONDITIONS: Several error conditions may be recoverable if the appropriate err clause is included in the WRITE statement or on a referenced EXIT statement.

The CONV condition occurs if a value cannot be converted to the specified format, or if the record being added is larger than the record length for the file.

The DUPKEY or DUPREC conditions occur if the key or record number specifies a record which already exists.

The SOFLOW condition indicates the occurrence of a string overflow.

The EOF condition occurs if there is not enough room to add the record.

The IOERR condition is caused by hardware malfunction or other conditions which prevent the writing of the record.

The error clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 84.

Example

```
100 WRITE #5, USING 110, REC=1234: FLDA,FLDB,FLDC&
  & IOERR 500, EOF 600
110 FORM X 10,N5,X 10,N5,X 10,N5
```

In this example, a record with relative position 1234 is added to the relative file associated with file reference number 5. The record is 45 positions in length, with three numeric values each preceded by ten blanks. If an end of file occurs, control would pass to line 600. An IOERR exception would result in control being transferred to line 500.

In interactive mode, a user can enter a statement from the terminal without a line number. In this case, the statement is translated and processed immediately rather than stored away for later execution.

Immediate statements can be used in "desk calculator" mode of execution.

Immediate statements are also useful for program debugging. During a breakpoint, when the program is executing in debugging mode, any immediate statement can be entered. Thus, through immediate statements, the user can inspect the values of program variables at intermediate points during execution.

Not all statements may be executed immediately, and those that may often have special semantics or restricted syntax. This section identifies which statements can be used immediately and discusses general rules for the use of immediate statements.

Specific rules for each statement are given under the heading "Immediate Execution" as part of the individual statement definitions in the section "Statement Descriptions" on page 88.

Statements that can be used immediately are:

DEBUG
DECIMAL
DIM
INTEGER
LET
MAT
PRINT
OPTION
RANDOMIZE
STOP
TRACE

You can execute an immediate statement anytime your session is in command mode; command mode is signified by an asterisk prompt on the terminal screen.

The user can continue immediate statements across more than one input line; however, immediate statements must be entered one at a time. Multiple statements per line are not accepted.

VARIABLES AND ARRAYS AND IMMEDIATE STATEMENTS

When a program is stopped at a breakpoint, the user can issue immediate statements to display or change the values of the program's variables and arrays.

Immediate statements may also use variables and arrays other than those specifically stated in the program. In fact, a program may be either suspended, ended, or not present (that is, the workspace is empty) when an immediate statement is processed. These "immediate variables" are created (attributes attached and storage allocated) according to immediate declarations and immediate options.

Example

```
OPTION BASE 1
INTEGER A
DIM A(20)
```

These three immediate statements create the immediate, 20-element, integer, array A.

The scope of an immediate variable is also the containing program unit, which is defined as:

- The main program if the variable is created at any time other than when at a subprogram breakpoint. This includes when you are not at a breakpoint (the program is not running) or when there is no program in the workspace.
- An instance of a subprogram if the variable is defined while at a breakpoint in the subprogram. Note that in the case of recursive calls, different instances of the same subprogram may contain different immediate variables.

The only program variables that may be accessed at any given breakpoint are the variables belonging to the containing program unit—main program or subprogram.

For example, assume the main program contained the variables ALPHA and BETA. The main program CALLs a subprogram named SUB1 which also contained a variable named BETA. The variable printed by "PRINT BETA" would depend upon where execution was stopped, in main or SUB1. Also, at a breakpoint in SUB1, "PRINT ALPHA" would not access main's ALPHA, but would create an immediate ALPHA (and print zero, the default initial value).

Subprogram variables (immediate and program) cease to exist when the subprogram is exited. Main program variables continue to exist after the program completes. In other words, after a program has been executed, immediate statements may be used to inspect the final values of the program variables.

Execution of any command or editing operation which changes the program in the workspace causes IBM BASIC to drop all immediate and program variables. In addition, the COMPILE, INITIALIZE, and RUN commands cause all variables to be dropped (although RUN causes another generation of program variables to come into existence).

The DROP command explicitly removes immediate and program variables. This may be necessary if the user wishes to attach new attributes to an existing variable name. The old attributes must be detached before the processor will accept a new declaration.

IMMEDIATE TYPE AND DIMENSIONS

Immediate DECIMAL and INTEGER statements set the type of immediate variables, and immediate DIM statements give dimensions for immediate arrays and establish maximum string lengths for immediate character variables. These statements have the same syntax rules as when they are used in a program, but they behave slightly differently in immediate mode.

Immediate declarations have no effect on program variables (variables used by the program). The attributes of the program variables are completely determined by the program.

An immediate DECIMAL or INTEGER statement does not "create" variables. It simply records that the specified identifiers and/or first-letters are assigned a particular type. It is not until a subsequent immediate statement (for example, LET) uses an identifier in a context requiring a variable or array that the type information is used. Because of this delayed action, immediate type statements (DECIMAL or INTEGER) may contradict previous immediate type statements and, if at a breakpoint, program type statements. However, the new type declarations must not explicitly (that is, by name) contradict a variable or array that has already been created, either by the program or by previous immediate statements.

Example

```
INTEGER ALPHA
LET ALPHA = 5
DECIMAL ALPHA
```

is an error; the LET statement created ALPHA with integer type.

```
INTEGER ALPHA
DECIMAL ALPHA
LET ALPHA = 5
```

is acceptable because ALPHA has not been created when the DECIMAL statement is entered.

The first letter typing stated in immediate DECIMAL and INTEGER statements may contradict both previous immediate first-letter typing and program first-letter typing. The immediate first-letter typing applies only to immediate variables and arrays created subsequently.

Example

The program contains

```
100 INTEGER RED,(A-C)
110 RED=1
120 BLUE=2
```

when stopped at a breakpoint,

DECIMAL RED is an error

DECIMAL (B-D) is OK and has no effect on the integer program variable BLUE.

An immediate DIM statement causes arrays and character variables to be created. Type declarations must have preceded the DIM statement.

Example

```
DIM GREEN (50)
INTEGER GREEN
```

is an error. GREEN is assigned decimal type (the default) as part of the DIM statement.

IMMEDIATE STATEMENT EXCEPTIONS

All exceptions generated by immediate statements are handled according to the default actions (the SYSTEM actions in the ON Condition statement). See "ON Condition Statement" on page 203.

ON Condition statements in the workspace have no effect upon immediate statements. Nor can immediate PRINT statements contain "err" clauses (which refer to lines in the workspace).

EDITING WITH LINE NUMBERS

One of the major features of IBM BASIC is the ease with which the user can enter, update, and execute a program.

THE WORKSPACE

BASIC maintains the current program in an area called the workspace—an area of storage reserved for a user's exclusive use. The workspace can be given a name by using one of several commands: FETCH, INITIALIZE, LOAD, or RENAME.

When you create a program it is entered, line by line, into the workspace. When you log on, your workspace is empty.

You can enter program lines in your workspace either directly from the terminal keyboard or, by using commands, from files in your library. Each time you want to change a program, you must load your program into your workspace so that those changes can be made. You ask for the file that contains your program to be put into your workspace by using the LOAD command.

```
LOAD file-name
```

When you have made all of the changes you feel are necessary, you can save the new version of the program in a file by using the SAVE command. The workspace is the collection of information which completely describes the program you are currently editing and/or executing. This includes the program itself, the data being acted upon, and other information concerning the status of the program.

ENTERING PROGRAM LINES FROM THE TERMINAL

IBM BASIC indicates that a line entry is expected by displaying an asterisk (*) on the terminal. Every program line begins with a line number. The presence of a line number on an input line identifies the line as a program statement rather than a command or immediate statement. Program statements require line numbers; commands and immediate statements do not begin with numbers.

Program lines may be entered in any sequence. IBM BASIC automatically accumulates them in the workspace and sorts them by line number.

Every program line begins with a line number; however each program line may be composed of more than one physical line (the original line plus its continuations). Each such physical line is a separate record.

If you enter a continued record (ending with a continued ampersand that is not part of a REM statement), IBM BASIC prompts for a continuation record by displaying the leading continuation ampersand.

REPLACING AND DELETING INDIVIDUAL LINES

Program lines already entered may be replaced by reentering the line using the same line number.

Individual program lines already entered may be deleted by entering only the line number.

EDITING CONTINUATION RECORDS

A continuation record within a line can be referred to by its record number. For example, if line 200 required four records to complete, they would be numbered 200.0, 200.1, 200.2, and 200.3. The number to the right of the decimal point indicates the continuation record number.

This notation is not generally available in commands, such as DELETE or LIST, which refer to entire lines, but may be used to delete, replace, insert, and, in the second format of the CHANGE command, change records as documented below. Continuation numbers are never displayed.

DELETING CONTINUATION RECORDS

A continuation record may be deleted by entering:

```
line-number.record-number
```

The record number must be greater than or equal to 1, and must designate an existing record. If not, an error message is displayed and the workspace is not changed.

Example

```
110.1
```

deletes the first continuation record of line 110.

REPLACING RECORDS

A record may be replaced by entering:

```
line-number.record-number new-data
```

The record number must be greater than or equal to 0, and must designate an existing record. If not, an error message is displayed and the workspace is not changed.

If the last record of a line is replaced and the new record ends with an ampersand and the line is syntactically correct up to that point, the user is prompted for another continuation record with an ampersand. This allows the user to extend a line without creating meaningless error messages.

Example 1

```
150.3&, ORGANIZATION RELATIVE &
```

replaces the third continuation record of line 150 with

```
&,ORGANIZATION RELATIVE &
```

and prompts with 150.4&

Example 2

```
120.0 A = 2**B&
```

replaces the line number record with

```
120 A = 2**B&
```

In each example, IBM BASIC then prompts for the next line with an ampersand.

INSERTING CONTINUATION RECORDS

A new continuation record may be inserted by entering the initial line segment number, a + sign, and the segment number, as follows:

```
line-number+record-number new-data
```

The record-number must be greater than or equal to 1. If it is not, an error message is displayed and the workspace is not changed.

If the record number is greater than the existing number of continuation records, then the new record is added after the last record in the line.

If the inserted record is the last record of a line and it ends with an ampersand and the line is syntactically correct to that point, the user is prompted for another continuation record with an ampersand.

Example

```
150+3&, TYPE NATIVE &
```

inserts a record

```
&, TYPE NATIVE &
```

after the second continuation record of line 150 and prompts with an ampersand. The inserted record is then the third continuation record.

INSERTING CONTINUATION RECORDS

A new continuation record may be inserted by entering the initial line segment number, a + sign, and the segment number, as follows:

```
line-number+record-number new-data
```

The record-number must be greater than or equal to 1. If it is not, an error message is displayed and the workspace is not changed.

If the record number is greater than the existing number of continuation records, then the new record is added after the last record in the line.

If the inserted record is the last record of a line and it ends with an ampersand and the line is syntactically correct to that point, the user is prompted for another continuation record with an ampersand.

Example

```
150+3&, TYPE NATIVE &
```

inserts a record

```
&, TYPE NATIVE &
```

after the second continuation record of line 150 and prompts with an ampersand. The inserted record is then the third continuation record.

IBM BASIC COMMANDS

This section contains individual discussions of each interactive command. The commands are presented in alphabetic order.

ABBREVIATION OF COMMANDS

All commands may be abbreviated, but only if the abbreviation is unique. For example, the commands QUERY and QUIT can be abbreviated as follows:

```
QUERY     QUIT
QUER      QUI
QUE
```

If either is abbreviated further, BASIC cannot determine which command is meant.

Figure 41 lists each command with its shortest allowable abbreviation.

The command description formats in the following section show the minimum abbreviation for each command.

Command	Abbreviation
AUTO	AU
BREAK	BR
CHANGE	CH
COMPILE	COM
COPY	COP
DELETE	DE
DROP	DR
EXTRACT	EX
FETCH	FE
FIND	FI
GO	GO
HELP	HE
INITIALIZE	IN
LIST	LI
LOAD	LO
MERGE	ME
PURGE	PU
QUERY	QUE
QUIT	QUI
RENAME	RENA
RENUMBER	RENU
RUN	RU
SAVE	SA
SET LOG	SE LOG
SET MSG	SE MSG
STORE	ST
SYSTEM	SY

Note: The minimum abbreviation is two characters

Figure 41. IBM BASIC Commands—Minimum Abbreviations

CURRENT LINE

When one or more program lines exist in your workspace, one of the lines is considered the current line. The current line is considered to be the point in the program at which you are currently editing. It is used as the implicit operand for one form of the CHANGE command and as a reference point for scrolling with the LIST command.

Usually the current line is the last line entered. However, commands that perform editing functions may modify the setting of the current line. Those commands that change the current line are noted in the following discussions.

AUTO COMMAND

The AUTO command puts a terminal session into program line entry mode.

Format

AUTO [line-number][STEP increment]

Minimum: AU

Where:

line-number
indicates the starting line number. The default is 100.

increment
is a nonzero, positive integer less than 9999999. The default is 10.

Description

In program line entry mode, the processor presumes the user is entering consecutive lines of a program and prompts with line numbers so that all the user need do is enter BASIC statements.

The first prompted line number is determined by the following rules:

- If line-number is specified, the first prompted line is for line-number.
- If line-number is not specified and the workspace is empty, the first prompted line is for 100.
- If line-number is not specified and the workspace is not empty, the first prompted line is equal to the highest line number in the workspace plus the increment (or 10 if the increment is not specified).

Subsequent line numbers are derived by adding the STEP increment, if specified, or 10, the default, to the previous automatically generated line number.

To terminate automatic line number prompting, enter the null line (press the ENTER key after a new line number is displayed). The null line does not become part of the program.

Line number prompting is also terminated by the following error conditions:

- The next line number to be prompted would be equal to an existing line number in the workspace.
- An existing line number is greater than the last prompted line number and less than the next prompted line number.
- The line-by-line syntax checker detects an error (the associated syntax error message will be displayed).
- The next line number to be prompted would exceed the maximum line number (9999999).

As lines are entered after the line number prompts, the most recent becomes the current line.

Example 1

```
AUTO 300 STEP 5
```

starts prompting with 300 and increments by 5.

Example 2

```
AUTO
```

starts prompting at 100 and increments by 10, (if the workspace is empty).

Example 3

If the workspace currently contains the lines

```
100 A=B
110 C=D
```

then

```
AUTO
```

starts prompting at line 120 and increments by 10.

BREAK COMMAND

The BREAK command sets, removes, and lists breakpoints within a program.

```
Format 1
    BREAK [ON] line-1 [,line-2]...
Format 2
    BREAK OFF [line-1 [,line-2]...]
Format 3
    BREAK?
Format 4
    BREAK ?
Minimum:  BR
```

Where:

line-n
specifies an actual line number.

Description

A BREAK ON command causes execution to halt just prior to the line specified. The BREAK ON command sets breakpoints at the indicated line number.

The BREAK OFF command removes breakpoints. If no line numbers are specified, all breakpoints are removed. If one or more line numbers are specified, only those breakpoints are removed.

BREAK? or BREAK ? lists all line numbers where breakpoints are currently set. If no breakpoints are set, the message "NO BREAKPOINTS" is displayed.

When a breakpoint is encountered during execution, execution is suspended and a message is displayed indicating the line number. Execution is suspended before the line is executed. You may then use immediate statements and commands to inspect the values of variables, set additional breakpoints, etc., before continuing execution.

Execution can be resumed, if nothing is done while at the breakpoint, by pressing the ENTER key or by issuing the GO command.

If commands that do not alter the workspace program or immediate statements have been executed, the GO command must be used to resume execution.

If a command that alters the workspace program or a DROP command that drops a program variable is executed, then execution of the workspace program cannot be resumed.

A breakpoint remains in effect if:

- The line associated with the breakpoint is replaced by line editing.

- The line associated with the breakpoint is renumbered (RENUMBER command). It is the program line that has the breakpoint, not the line number. The line numbers are used to help point to the program line.

A breakpoint is automatically deleted if:

- The line associated with the breakpoint is deleted.
- The MERGE command replaces the line associated with the breakpoint.

All breakpoints are deleted as part of INITIALIZE, LOAD, and FETCH commands.

Example

If a breakpoint is set on line 100 as follows:

- BREAK ON 100
- Line 100 must exist at the time the breakpoint is set.
- If line 100 is deleted, the breakpoint is removed. If a new line 100 is added later, line 100 does not have a breakpoint set.
- If you replace line 100 by typing "100" followed by a new statement, line 100 still has a breakpoint set.
- If the MERGE command replaces line 100, line 100 no longer has a break point set.
- If RENUMBER changes line 100 to line 150, line 150 still has a breakpoint set, but not the new line 100, if any.

The current line setting remains unchanged for a BREAK command.

Example 1

```
BREAK ON 200,300
```

sets breakpoints at lines 200 and 300.

Example 2

```
BREAK OFF 200
```

removes a breakpoint previously set for line 200. Note that the break previously set for line 300 (example 1) remains in effect until it is set OFF also.

CHANGE COMMAND—FORMAT 1

The CHANGE command changes character strings within the statements in your workspace. There are two formats of this command. This section discusses the first format.

Format 1

```
CHANGE [start-line [TO end-line]]
       delim old-string
       delim new-string
       [delim [A[LL]]]
```

Minimum: CH

Where:

start-line and end-line
specify the scope of the command. They may also be specified FIRST or F to refer to the lowest line number and LAST or L to refer to the highest line number.

delim
is a delimiter of the string. It must be a special character (other than 0-9 or A-Z) including the space character such that:

- delim is not contained in either old-string or new-string.
- delim may be a space (or series of spaces) only if all following conditions are true:
  - old-string does not start with a special character or a digit (when start-line is omitted)
  - old-string is not FIRST, F, LAST, L (when start-line is omitted)
  - old-string is not TO (when end-line is omitted but start-line is specified).

old-string
is the character string to be changed.

new-string
is the new character string to replace the old string.

A or ALL
indicates that all occurrences of old-string are to be replaced by new-string.

Description

The CHANGE command specifies lines in the workspace to be changed. All occurrences of old-string can be replaced by new-string (the ALL option), or only the first occurrence in each line.

If no line numbers are given (no start-line or end-line), the Current Line is the line to be changed.

If only start-line is specified, it must be an actual line number of your program, and only that line is changed.

If both start-line and end-line are specified they need not be actual line numbers but must bracket at least one actual line; all lines between start-line and end-line, inclusive, are changed.

If ALL is specified, all occurrences of old-string in each line of the range are replaced by new-string. If ALL is not specified, only the first occurrence in each line of the range is replaced.

The leading line numbers on each line are not considered part of the line to be searched. Therefore you cannot change the line number of a line, only the statements within the line.

Old-string cannot overlap continuation records.

Though the IBM BASIC language, except in character strings, does not discern between upper and lower case, the CHANGE command finds only new strings which exactly match old-strings.

If new-string is a null string (two successive delimiters), the effect is deletion of old-string.

If old-string is a null string, new-string is inserted immediately after the blank character following the line number.

If old-string cannot be located,

```
"STRING NOT FOUND"
```

is displayed.

When any change occurs, the new line is displayed. If more than one line is being changed, all changed lines are displayed, and at the end of all the changes, a count of the total number of lines changed is displayed on your terminal.

Whenever a line is changed, the syntax of the new line is checked. Consequently, error messages may appear along with the new line.

If a string replacement would result in a record longer than the maximum allowed, the replacement is not made, a message is displayed, and the CHANGE process ends at the affected line.

The leading and trailing ampersands on the records of a continued line may be changed. This may result in syntax errors. Note that all the records between one line number and the next are associated with the first of the two line numbers, irrespective of the records' contents. This means you can inadvertently remove a leading or trailing continuation ampersand and still not lose the following continuation records.

The current line is reset to the last line accessed by CHANGE, or the last line in the range if a range of line numbers is specified.

Example 1

```
CHANGE /DAYS/WEEKS/
```

The first occurrence of DAYS, on the current line, is changed to WEEKS. If DAYS occurs again, either in the same line, or in any other line, it remains unchanged.

Example 2

```
CHANGE 100 /DAYS/WEEKS/ALL
```

Every occurrence of DAYS, on line 100, is changed to WEEKS.

Example 3

```
CHANGE 100 TO 200/DAYS/WEEKS/ALL
```

All occurrences of DAYS, between lines 100 and 200 inclusive, are changed to WEEKS.

Example 4

```
CHANGE 100/DAYS//ALL
```

Delete all occurrences of DAYS found on line 100. The delete is specified by providing two consecutive delimiters, (the null string) for the new string.

Example 5

```
CHANGE //DAYS/
```

Inserts the word DAYS following the space after the line number of the current line.

Example 6

```
CHANGE FIRST TO LAST/DAYS//ALL
```

Deletes all occurrences of DAYS from the workspace.

Example 7

```
CHANGE F TO 400/DAYS/WEEKS/
```

Change the first occurrence of DAYS to WEEKS in each line from the first line in the workspace through the line numbered 400.

Example 8

```
CHANGE 400 TO LAST/DAYS/WEEKS/A
```

Change every occurrence of DAYS to WEEKS found from line 400 through the last line in the workspace.

Example 9

```
CH OLD NEW
```

Change the first occurrence of OLD to NEW in the current line.

CHANGE COMMAND—FORMAT 2

The CHANGE command changes character strings within the statements in your workspace. There are two formats of this command. This section discusses the second format, which may be used only with a 327X type terminal.

Format 2

```
CHANGE [line-number [.record number]]
```

Minimum: CH

Where:

line-number
identifies an existing line in the workspace. It is an unsigned integer.

record number
identifies a particular record of a multirecord line (a continued line). It is an unsigned integer.

Description

The second form of the CHANGE command does not perform string replacement. It displays a particular record in the 327X terminal input area that can then be modified and reentered.

If you enter a line number, that line is displayed in the input area of the screen.

If you enter no line number, the current line is displayed in the input area of the screen.

The record number is needed only if you are dealing with continued lines and want to change a particular continuation record. Individual continuation records of a line are numbered starting with zero. Thus, line 250 itself can be considered line 250.0, and the second continuation record (the third record of line 250) is 250.2.

As discussed for format 1 of CHANGE, the continuation ampersands may be altered without loss of the record.

The line number of a changed line must not be changed. If it does not agree, the modified record is rejected with an error message.

An error message is displayed if the terminal (for example, a hard copy terminal) does not support the editing facilities.

The current line is reset to the line accessed by the CHANGE command.

Example 1

If your program contains the line

```
200 LET DAYS = 52
```

you can recall this line to the screen with

```
CH 200
```

and then, using the terminal's editing keys, change it to

```
200 LET WEEKS = 52
```

and reenter the line in your workspace by pressing ENTER.

Example 2

If your program contains the line

```
100 OPEN #5: NAME "MYFILE" &
  &  ,ACCESS INPUT         &
  &  ,TYPE NATIVE          &
  &  ,ORGANIZATION KEYED   &
  &  ,RECORDS FIXED        &
  &   EXIT 9000
```

The third record (the one containing TYPE) can be brought to the screen for editing by

```
CHANGE 100.2
```

COMPILE COMMAND

The COMPILE command compiles the program currently located in the workspace.

Format

```
COMPILE [OBJECT (file-spec-1)]
        [OUT (file-spec-2)]
        [OPTIONS (option-list)]
```

Minimum: COM

Where:

file-spec-1 and file-spec-2
are file names.

option-list
is a list of compiler option keywords. The choices are given below.

The parameters OBJECT, OUT, and OPTIONS may appear in any order.

Description

Execution of the COMPILE command is almost the same as the execution of the batch IBM BASIC compiler, the only difference being that COMPILE expects the source program in your workspace instead of in a file.

OBJECT CLAUSE: The OBJECT clause specifies the file on which the object text is to be placed. If omitted, the object text is placed in your library under the name currently associated with the workspace and a file type of TEXT.

OUT CLAUSE: The OUT clause directs the listing to a file. If omitted, the listing is placed in your library under the name currently associated with the workspace and a file type of LISTING.

If both the OBJECT and OUT clause are omitted and the workspace does not currently have a name, you are prompted for a name. You must enter a name or cancel the command by entering a null line.

COMPILER OPTIONS: The following compiler options are available. Each option is a keyword. The keywords must be separated by blanks or commas.

Each option has a default; compile any program without specifying any OPTIONS to discover the defaults in force for your organization.

SOURCE - NOSOURCE
Specifies whether or not the source program listing is to be written. This listing includes diagnostic error messages. Most errors are indicated by listing the statement in its original form with the erroneous phrases or characters undermarked, followed by messages indicating the error type. In addition, errors involving the flow of control, for example, branches to undefined statement numbers, are listed at the end of the program.

OBJECT - NOOBJECT
Specifies whether or not the object text is to be written.

MAP - NOMAP
Specifies whether or not to produce an allocation map, listing all the variables and subprograms used in each program unit.

The listing is in three parts: COMMON variables, local variables, and subprograms. Within each part, identifiers are listed alphabetically with type, location, and (arrays only) number of declared dimensions.

XREF - NOXREF
Specifies whether or not to produce a cross-reference listing for variables, referenced line numbers, and referenced statement labels.

This listing is in three parts in the following order:

1. Line numbers in numeric order
2. Line labels in alphabetic order
3. Variables, arrays, user-defined functions, and intrinsic functions in alphabetic order

With each number, label, or variable listed are the line numbers containing reference(s) to the item.

References to a statement label are followed by a colon.

References that may alter the value of the variable are followed by an asterisk.

References which are declarations, for example, in an INTEGER or COMMON statement, are followed by a slash.

LIST - NOLIST
Specifies whether or not to produce a listing of the object module, consisting of assembler mnemonics and symbols. If the SOURCE option is also specified, instructions are listed directly after the source statement lines to which they apply.

FLAG (I|W|E|S)
Specifies the level of diagnostic messages to be written.

FLAG(I) indicates that information messages, warning messages, error messages, and severe error messages are to be printed.

FLAG(W) indicates that warning messages, error messages, and severe error messages are to be printed.

FLAG(E) indicates that only error messages and severe error messages are to be printed.

FLAG(S) indicates that only severe error messages are to be printed.

FIPS - NOFIPS
Specifies whether or not to produce a diagnostic warning for any statement found not to conform to the ANSI Minimal Standard BASIC syntax.

These messages are informational and are printed only if FLAG(I) is set, either explicitly or by default.

SPREC - LPREC
Specifies the maximum number of significant digits to be written by the PRINT statement (without the USING clause) when printing decimal values. SPREC specifies a precision of 6 digits. LPREC specifies a precision of 12 digits.

Example 1

```
COMPILE OBJECT (MYOBJECT)
```

compiles the source code in your workspace and writes it to a file named MYOBJECT.

Example 2

```
COMPILE OBJECT (MYOBJECT) OUT (LISTFILE)
```

compiles the source code in your workspace, writes it to a file named MYOBJECT, and places the generated listing on the file named LISTFILE.

COPY COMMAND

The COPY command causes one or more lines in the workspace to be duplicated.

Format

```
COPY start-line [TO end-line] AT line-num
     [STEP increment]
```

Minimum: COP

Where:

start-line
may be either a line number, the keyword F[IRST], or the keyword L[AST].

end-line
may be either a line number or the keyword L[AST]. End-line must be greater than or equal to start-line.

line-num
is a line number.

increment
a nonzero, positive integer less than 9999999. The default increment is 1.

Description

The COPY command specifies the range of lines to be duplicated and the new line numbers to which they are assigned.

If only start-line is specified, then a single program line is copied and start-line must be an actual line number.

If a range, start-line TO end-line, is specified, all program lines within the range are copied. If there are no lines within the range, an error message is displayed.

If the COPY command would cause the new block of numbers to overlay or duplicate existing line numbers, an error message is displayed and the command is ignored.

If renumbering a reference causes a record to exceed the maximum length, an error message is displayed and the command is ignored.

The current line is set to the last line copied.

AT CLAUSE: Line numbers are assigned to the copied lines by assigning line-num to the first copied line and subsequent copied lines are assigned line numbers by adding the STEP increment to the previously assigned line number.

STEP OPTION: If the STEP option is specified, subsequent copied lines are numbered based on the increment set by this parameter.

If within the block of copied lines, a statement refers to a line within the block, that reference is adjusted to match the new numbers. References from outside the block are not modified.

Example 1

```
COPY 120 TO 170 AT 500 STEP 5
```

Copies the lines between 120 and 170, inclusively. The new lines will start at 500 and be incremented by 5.

DELETE COMMAND

The DELETE command deletes the specified line, or group of lines, from the workspace.

Format

```
DELETE start-line [TO end-line]
       [,start-line [TO end-line]]...
```

Minimum: DE

Where:

start-line
may be either a line number, the keyword F[IRST], or the keyword L[AST].

end-line
may be either a line number or the keyword L[AST]. End-line must be greater than or equal to start-line.

Description

When only start-line is specified, then start-line is deleted. Start-line, in this case, must be an actual line number.

If a range, start-line TO end-line is specified then all lines within the range, inclusive, are deleted.

A message for each range of lines and/or single line deleted will be displayed upon completion of the command.

If a specified range or single line number does not exist (that is, there are no lines within the range), a message is displayed and the DELETE command is ignored.

The current line is set to the next line after the largest line number deleted. If such a line does not exist, the current line is set to the last line in the workspace.

Example 1

```
DELETE FIRST TO 130
```

deletes all lines up to and including 130.

Example 2

```
DELETE 100, 200 TO LAST
```

deletes line 100 and all lines after 199.

DROP Command

DROP COMMAND

The DROP command erases the specified variables.

Format

DROP [identifier [,identifier]...]

Minimum: DR

Where:

identifier
can be any valid variable or array name.

Description

The DROP command erases either immediate or program variables, without resetting the entire workspace. This will also free the variable name so that it may be reused. If DROP is used without a variable list, all currently active variables are forgotten.

Erasing a program variable while execution is halted at a breakpoint prohibits program continuation. The program must be restarted at the beginning.

After the identifier is dropped, it may be redeclared in an immediate statement, for example, to change the type or number of dimensions.

Example

```
DROP OLDVAR
```

Erases the variable OLDVAR, leaving the rest of the workspace unchanged.

EXTRACT COMMAND

The EXTRACT command removes all lines, other than those specified, from the workspace.

Format

```
EXTRACT start-line [TO end-line]
        [,start-line [TO end-line]]...
```

Minimum: EX

Where:

start-line
may be either a line number, the keyword F[IRST], or the keyword L[AST].

end-line
may be either a line number or the keyword L[AST]. End-line must be greater than or equal to start-line.

Description

If only start-line is specified, all program lines are deleted from the workspace except start-line, and start-line must be an actual line number.

If a range, start-line TO end-line, is specified, all program lines, except those within the specified range, are deleted from the workspace. Start-line and end-line need not be actual line numbers.

A message for each successful range of lines and/or single line extracted will be displayed upon completion of the command.

The current line is set to last program line in the workspace.

If a specified range or single line number does not exist (that is, there are no lines within the range), a message is displayed and the EXTRACT command is rejected.

Example 1

```
EXTRACT 115 TO 120
```

Deletes all lines before 115 and all lines after 120 from the program, keeping lines 115 through 120.

Example 2

```
EXTRACT 115 TO 120, 125 TO 130
```

Deletes lines FIRST through 114, 121 through 124, and 131 through LAST, keeping lines 115 through 120 and 125 through 130.

Example 3

```
EXTRACT FIRST TO LAST
```

Deletes no lines. The workspace is unchanged, except that the current line is now the last line of the workspace.

FETCH Command

FETCH COMMAND

The FETCH command restores programs to the workspace.

Format

```
FETCH file-spec
```

Minimum: FE

Where:

file-spec
is a file name.

Description

FETCH is used in conjunction with the STORE command. You can save the contents of the workspace to a file by using the STORE command, and later issue a FETCH command, causing your workspace to be restored to the exact state it had prior to the STORE.

Unlike the LOAD command which scans and translates to internal text, FETCH merely brings the program into storage. Individual syntax errors are not reported, even if they exist. However, if you attempt to run a program and it contains errors, you are notified.

The program in the workspace previously, if any, is cleared as part of FETCH. The file-spec becomes the name associated with the workspace.

FETCH sets the current line to the last line in the workspace.

Example

```
STORE SAVEDATA
.
.
.
FETCH SAVEDATA
.
.
.
```

(See "STORE Command" on page 315.)

FIND COMMAND

The FIND command locates character strings within a program line, or block of lines and displays the line number on the terminal.

Format

```
FIND [start-line [TO end-line]]
     delim string [delim[A[LL]]]
```

Minimum: FI

Where:

start-line
may be either a line number, the keyword F[IRST], or the keyword L[AST].

end-line
may be either a line number or the keyword L[AST]. End-line must be greater than or equal to start-line.

delim
is a delimiter of the string. It must be a special character (other than 0-9 or A-Z), including the space character, such that:

1. delim is not contained in string.
2. delim may be a space (or series of spaces) only if string does not start with a special character or a digit (when start-line is omitted) and string is not FIRST, F, LAST, L (when start-line is omitted) and string is not TO (when end-line is omitted but start-line is specified).

string
is an exact representation of the character string you are trying to find.

Description

The range of the FIND command is determined by the following rules:

- If neither start-line nor end-line is specified, the range is assumed to be from FIRST to LAST.
- If only start-line is specified, the range is from start-line to LAST.
- If both start-line and end-line are specified, the range is from start-line to end-line.

If A[LL] is specified, all lines in the range that contain string are displayed.

If A[LL] is not specified, the first line in the range containing string is displayed.

The value of the current line at the completion of a FIND command depends upon two conditions, the ALL option, and whether or not the string is found:

- If ALL is specified, the current line is set to the last line in the range, either end-line or LAST.

- If ALL is not specified and the string is not found, the current line is set to the last line in the range.
- If ALL is not specified and the string is found, the current line is set to the line in which the string was found.

If the string is not found, the following message is displayed:

"STRING NOT FOUND"

The leading line number and following blank are not considered as part of the line to be searched.

Example 1

```
FIND /THE END/ALL
```

Find and display every line with the occurrence of the string THE END.

Example 2

```
FIND 100 TO 150/THE END/
```

Find the first occurrence of THE END after line 99 and before line 151, and display that line on the screen.

Example 3

```
FIND /THE END/
```

Find and display the first occurrence of THE END.

GO COMMAND

The GO command restarts a program which has been temporarily halted.

Format 1

```
GO [{line-number|END}][STEP]
```

Format 2

```
GOTO line-number
```

Minimum: GO

Where:

line-number
corresponds to an actual line number in the program.

Description

The GO or GOTO command resumes execution.

Execution is suspended in a number of ways:

- The next line to be executed has been designated as a breakpoint in a BREAK command
- Executing a BREAK statement
- Executing a PAUSE statement
- Executing an ATTENTION interrupt

If no line number is specified, execution will resume at the first statement following the last program statement executed. Execution sequence may be altered by specifying a different line number with the command.

If you specify a line number it must be within the current program unit. You cannot use the GO or GO TO commands to enter into or exit from a subprogram. You must also take care that you do not request an invalid entry or exit of:

- DEF/FNEND blocks
- FOR/NEXT blocks
- DO/LOOP blocks
- IF/CASE/END IF blocks
- SELECT/CASE/END SELECT blocks

which could cause unpredictable results.

GO END terminates the program and closes all files. You can use this version of the command independently of the current program unit. The keyword STEP may be specified with END, but has no effect, that is, GO END STEP is the same as GO END.

The STEP keyword causes the program to execute one statement at a time, displaying the line number of the next statement expected to be executed. This is called a STEP prompt. Pressing the ENTER key

keeps the program in the STEP mode, executing one statement per entry.

Exit from STEP mode can be done in one of these ways:

- A response to the step prompt with anything other than the ENTER key, for example, with another command
- Program execution of a STOP, END, or BREAK statement
- Program generation of any exception that causes program termination

In STEP mode, breakpoints set by the BREAK command remain in effect but do not stop STEP mode.

Example 1

```
GO
```

continues execution.

Example 2

```
GO 500 STEP
```

continues execution in STEP mode at line 500

Example 3

```
GOTO 200
```

continues execution at line 200

Example 4

```
GO END
```

terminates program execution.

HELP COMMAND

The HELP command displays information about IBM BASIC.

Format

HELP [request]

Minimum: HE

Where:

request
is the name of a HELP panel.

Description

The HELP command has two primary usages. The first as a tutorial device invokable any time you want a brief statement about a particular aspect of writing IBM BASIC programs or using system services. The second as a diagnostic aid to provide you with explicit information on how to resolve a processor-noted error.

THE HELP TREE: The information displayed by HELP can be thought of as a tree of panels. Each panel is either a menu (points to other panels) or is prose (a leaf of the tree). Each panel may have several pages, in which each page is one full terminal display. The tree contains panels describing all IBM BASIC commands, statements, intrinsic functions, and diagnostics, as well as tutorial discussions of how to use the interactive facilities and how to write programs.

HELP MODE: HELP is a separate mode of interactive IBM BASIC. Once you enter HELP mode (by executing a HELP command), you must use HELP actions. The usual commands, line number editing, and immediate statements are not accepted until you leave HELP mode (by means of the HELP "CAN" action).

ENTERING HELP MODE: The HELP command causes the contents of your screen to be saved (if you are using a display terminal) and the first page of a HELP panel to be displayed. You are then in HELP mode. Your original screen is restored when you exit HELP mode.

You may specify the name of the panel you wish to have displayed, or you may enter "HELP" with no panel name. In the latter case, the processor determines the panel to be displayed:

- If your entry of the HELP command was immediately preceded by a diagnostic message, IBM BASIC assumes that you desire more information concerning the diagnostic, and displays a panel with that information.
- If the HELP command was not preceded by a diagnostic, a panel describing how to use HELP is displayed.

THE HELP SCREEN: While in HELP mode, the top line and the bottom two lines of a display terminal screen have a fixed format.

The top line identifies:

- BASIC HELP
- The current panel name
- "PAGE n OF m"

The bottom line defines the correspondence between PF keys and HELP actions.

The next to the bottom line is the command line. It contains the prompt "===>" to the left. HELP action keywords and panel names are entered on this line.

HELP ACTIONS: Once in HELP mode, you may use HELP actions to view the information in the HELP tree. These actions can be invoked by the entry of keywords or, for those terminals which have them, PF keys. (See Figure 42 for the PF keys that HELP uses.)

Keyword	PF Key	Action
Panel name	none	Displays the first page of the named panel. If there is no panel for the name, an error message is displayed on the command line.
CAN	PF12	Returns control to IBM BASIC command mode. Exits HELP mode (cancel).
HLP	PF1	Displays a panel which explains how to use HELP.
PRV	PF3	Restores the page of the previous panel which was being displayed when the current panel was requested. If there was no previous panel, returns control to IBM BASIC command mode (cancel). Exits HELP mode.
SCF	PF8	Advances to the next page of the current panel (scroll forward). If there is no next page, an error message appears on the command line.
SCB	PF7	Goes back to the previous page of the current panel (scroll backward). If there is no previous page, an error message appears on the command line.
PRT	PF5	Prints all of the current panel. The listing goes to a file named (BASHELP) of type listing. A message appears on the command line when the action is completed.

Figure 42. HELP—PF Keys Used

PRINTING ALL OR PART OF HELP: The PRT keyword (or PF5) may be used to print the current panel. The listing goes to a file named (BASHELP) of type listing.

The HELP command PRT MESSAGES may be used to print all the messages which can be issued by interactive IBM BASIC. The messages are printed to a file named (BASHELP) of type listing.

The help command PRT ALL may be used to print all the panels. The panels are printed to a file named (BASHELP) of type listing.

HELP WITH HARD COPY TERMINALS: When you request a HELP panel on a hard copy terminal, all of the designated panel is printed (all pages). The SCF and SCB actions are ignored, because all pages are printed at the same time.

Example

```
HELP READ
```

displays a panel describing the READ statement.

INITIALIZE COMMAND

The INITIALIZE command closes all open files and clears the workspace.

Format

```
INITIALIZE   [file-spec]
```

Minimum: IN

Where:

file-spec
is a file name which may be supplied to rename the current workspace.

Description

The workspace is cleared of all statements and data. If a filename is specified, it becomes the name of the empty workspace, otherwise the workspace is unnamed.

The state of the FIPS/NOFIPS and FLAG options are reset to the system defaults (see immediate options). The workspace is cleared of all statements and data.

The current line setting is undefined.

Example

```
INITIALIZE MYPROG
```

The empty workspace is named MYPROG.

LIST COMMAND

The LIST command causes the text of the specified program lines to be displayed.

```
Format 1

     LIST [XREF] [OUT(file-spec)]
          [line-number-range[,line-number-range]...]

Format 2

    LIST scroll-spec [n]

Minimum:  LI
```

Where:

file-spec
is a file name.

line-number-range
specifies one or a range of line numbers in the form

```
start-line [TO end-line]
```

where start-line may be the keywords F[IRST] or L[AST]. End-line may be specified as L[AST].

scroll-spec
is the keyword FOR[WARD] or the keyword BACK[WARD].

n
is a positive, nonzero integer.

The XREF clause, OUT clause, and line number ranges may appear in any order.

Format 1 Description

The LIST command causes the text of the specified lines between start-line and end-line, inclusively, to be displayed. If no line numbers are specified, the entire workspace is displayed. If only start-line is specified, only that line number is displayed.

Start-line and end-line need not be actual line numbers in the workspace. If a request is made to list a nonexistent individual line or a range of line numbers which does not include any lines, an error message is displayed.

If the keyword XREF is specified, a cross-reference listing of the program in your workspace is displayed. The listing is in three parts:

1. Referenced line numbers in numeric sequence
2. Line labels in alphabetic sequence
3. Variables, arrays, user-defined functions, and intrinsic functions in alphabetic sequence

Each item displayed indicates all line number references to that item:

- References to the line label are marked by a colon.

- References that may alter the value of a variable are marked by an asterisk.
- References that are declarations, for example, in an INTEGER or COMMON statement, are marked by a slash.

If you request a cross-reference of lines lying in more than one program unit, each program unit is cross-referenced separately. For example, if your workspace contains a main program followed by a subprogram, the command LIST XREF first displays the main program and its cross-reference listing and then displays the subprogram and its cross-reference listing.

Normally the cross-reference listing is displayed at the terminal. The OUT (file-spec) option may be used to direct output to a file.

If the requested listing contains more lines than can be held on the screen, the screen is filled and a message appears, requesting you to continue or terminate the scrolling.

If you respond with a null line (press the ENTER key), the listing continues to scroll until either the screen is full of new lines or the LIST command completes execution.

This format of the LIST command sets the current line to the last line listed.

To terminate the LIST command, respond with the attention interrupt.

Example

```
LIST 110 TO 120
```

displays lines 110 through 120 of your program on the terminal.

Format 2 Description

Scroll-spec is specified as either of the keywords BACK[WARD] or FOR[WARD]. Your program will be scrolled backward or forward from the current line in pages of lines specified by n. If n is omitted, 1 is assumed. A page of lines is the number of lines it takes to fill your terminal screen.

This format of the LIST command sets the current line to the last line listed.

Example

```
LIST FORWARD 2
```

Moves ahead two complete screens of program lines. In other words, one screen (or page) is skipped.

LOAD COMMAND

The LOAD command clears the workspace and loads a program from your library.

Format

LOAD file-spec

Minimum: LO

Where:

file-spec
is the file name of the program to be loaded into the workspace.

Description

When the LOAD command is executed, the workspace is cleared and the program in the file named in the command is loaded into the workspace.

Line-by-line syntax checking is performed. If any syntax errors are encountered, the lines in error and error messages are displayed. Syntax errors do not halt execution of the LOAD command.

After the file is loaded, the name of the program in the workspace is the file-spec used in the command.

The current line is set to the last line in the workspace.

If the specified file name is not a valid name or cannot be found, an error message is displayed. The workspace, the program name, and the current line setting remain unchanged.

LOAD is used in conjunction with the SAVE command. See "SAVE Command" on page 311.

Example

```
LOAD FILEUPDT
```

loads a program from a file named FILEUPDT; the name of the workspace is set to FILEUPDT.

MERGE COMMAND

The MERGE command inserts statements from files into the program currently in the workspace.

Format

```
MERGE file-spec [start-line][TO end-line]
      [AT line-number] [STEP increment]
      [(REPLACE[)]]
```

Minimum: ME

Where:

file-spec
is a file name.

start-line and end-line
are line numbers which specify a range of lines in the file to be merged. Start-line may be specified as F[IRST] or L[AST]. End-line may be specified as L[AST].

line-number
is a line number which is assigned to the first merged line, if specified.

increment
is a positive, nonzero integer less than 9999999.

The AT and STEP clauses may be interchanged.

Description

The MERGE command retrieves a sequence of statements from a file and inserts it starting at a specific line number in the workspace.

Start-line and/or end-line specify which lines from the file are to be merged; they need not be actual line numbers in the file. Start-line defaults to FIRST. End-line defaults to LAST. Thus, if you omit both start-line and end-line, the entire file (FIRST TO LAST) is merged. The merged lines are renumbered according to the AT and step parameters.

AT CLAUSE: The AT clause specifies where the merged lines should go in the workspace. It gives the line-number to be assigned to the first merged line. If the AT clause is omitted, it defaults to the first line number of the merged lines (the line number it has in the file).

STEP CLAUSE: The STEP clause indicates how the merged lines are renumbered. The increment between any two successive merged lines is as specified by the STEP clause, or, if you omit the STEP clause, the increment is that which existed between those two lines in the file.

If within the block of merged lines a statement refers to a line within the block, that reference will be renumbered, accordingly.

REPLACE CLAUSE: The REPLACE clause allows replacement of existing lines in the workspace. If any current lines are within the block of merged lines, and REPLACE is not specified, the merge does not take place. However, if REPLACE is specified, the merge takes place and the current program lines falling within the block of merged lines are deleted.

The current line is set to the last line merged.

Example 1

```
MERGE FILEA 350 TO 400 AT 350 STEP 5 (REPLACE)
```

Copies lines 350 to 400 from FILEA to the workspace renumbering them starting with 350 and incrementing by 5. If any existing lines are within the range of the merged lines, the old lines are deleted.

Example 2

If FILEA contains the lines

```
100 A = B
150 D = C
200 PRINT A,D
```

then

```
MERGE FILEA 100 TO 200
```

Inserts lines 100, 150 and 200 from FILEA in the program currently in the workspace, but does not renumber them and does not replace any lines in the workspace.

Example 3

```
MERGE FILEA 350 TO 400 AT 300
```

Insert lines 350 to 400 from FILEA (the merging program) into the workspace starting at line 300. The merged lines are renumbered from 300 on with the increment the same as in FILEA, but no statements in the merged section are replaced.

Example 4

If the contents of FILEA are:

```
.
.
.
500 KEY_ERRORS: EXIT DUPKEY 510, NOKEY 510
510             PRINT "KEY ERROR.LINE";LINE;",FILE#";FILENUM
520             CONTINUE
.
.
.
```

and the contents of the workspace are:

```
.
.
.
100 READ #11, USING 310: NAME$,ADDRESS$ EXIT KEY_ERRORS
210 FORM C18,C40
.
.
.
MERGE FILEA 500 to 520 AT 202 STEP 2
```

will change the workspace to:

```
.
.
.
100 READ #11, USING 210: NAME$,ADDRESS$ EXIT KEY_ERRORS
202 KEY_ERRORS: EXIT DUPKEY 204, NOKEY 204
204             PRINT "KEY ERROR.LINE";LINE;",FILE#";FILENUM
206             CONTINUE
210 FORM C18,C40
.
.
.
```

PURGE COMMAND

The PURGE command removes files from your library.

Format

PURGE file-spec

Minimum: PU

Where:

file-spec
is a filename.

Description

The PURGE command deletes all files named file-spec from your library. This may be more than one file if several files exist with the same filename but different file types. The types of files that may be removed are source, object, internal text, listing, or program data.

Example

```
PURGE YOURFILE
```

Removes the file YOURFILE from your program library and makes the space it occupied available for use by another file. The file is no longer available for processing, and must be rewritten if access to it is required.

QUERY Command

QUERY COMMAND

The QUERY command permits interrogation of your user files.

Format

```
QUERY [file-type] [file-spec-1]
      [OUT(file-spec-2)]
```

Minimum: QUE

Where:

file-type
is one of the following keywords:

```
ALL
FILE
PROG[RAM]
WORK[SPACE]
```

file-spec-1 and file-spec-2
are file names.

Description

The QUERY command obtains information about one or more files in your library.

FILE-TYPE: The file-type is a keyword indicating the kind of files about which you want information:

ALL means all types. This is the default if no keyword is given.

PROGRAM means source program files, those created by the SAVE command.

FILE means data files created by or for your programs.

WORKSPACE means program files, created by the STORE command.

FILE-SPEC-1: If you specify a particular file (file-spec-1) only, information regarding the requested types of files having that particular name is displayed. If no file-spec-1 is stated, all files of the requested types are displayed.

OUT (FILE-SPEC-2): Normally the requested data are displayed at the terminal but may be directed to a list file (file-spec-2) by using the OUT option.

Information on using the QUERY command is given in IBM BASIC Application Programming: System Services.

Example 1

```
QUERY PROGRAM
```

Obtains information about all source program files in your library.

Example 2

```
QUERY TEST
```

Obtains information about all files of all types with a filename of TEST in your library.

Example 3

```
QUERY WORK
```

Obtains information about all of the program files in your library (those created by a STORE command).

Example 4

```
QUERY
```

Obtains information about all files of all types in your library.

QUIT Command

QUIT COMMAND

The QUIT command ends the current BASIC session.

Format

QUIT

Minimum: QUI

Description

The QUIT command closes all open files, clears the workspace of all programs and data, and exits from BASIC.

If the program in the workspace should be saved, then a SAVE or STORE commands should be executed before the QUIT command.

Example

```
SAVE MYPROG
QUIT
```

or

```
STORE MYPROG
QUIT
```

RENAME COMMAND

The RENAME command either assigns a name to the program in your workspace or displays the current name associated with your workspace.

Format

RENAME [file-spec]

Minimum: RENA

Where:

file-spec
is a filename.

Description

The name of the workspace is changed to file-spec. If file-spec is not specified, the current name of the workspace is displayed.

If the workspace is unnamed, you are notified.

In response to the displayed name, a new name may be assigned by entering the new name. If a null line is entered, the name is unchanged.

If the workspace is named, and a file-spec is not specified, the following prompt is issued:

```
ENTER WORKSPACE NAME (NULL LINE FOR NO CHANGE):
```

The terminal user can then type in the new name.

When the new name is specified, the following message is issued:

```
'XXXXX' IS THE WORKSPACE NAME
```

(where XXXXX is the new name supplied).

If the workspace is unnamed, and a file-spec is not specified, this prompt message is issued. If the terminal user does not then enter a name, the following message is issued:

```
THE WORKSPACE DOES NOT HAVE A NAME
```

Example

```
RENAME NEWFILE
```

changes the name of the workspace to NEWFILE.

RENUMBER COMMAND

The RENUMBER command changes some or all of the existing line numbers of the program in the workspace.

Format

```
RENUMBER [start-line [TO end-line]] AT line-number]
         [STEP increment]
```

Minimum: RENU

Where:

start-line
may be a line number, the keyword F[IRST], or the keyword L[AST].

end-line
may be either a line number or the keyword L[AST]. End-line must be greater than or equal to start-line.

line-number
is a line number. The default is 100.

increment
is a nonzero, positive integer less than 9999999. The default increment is 10.

The AT and STEP clauses may be interchanged.

Description

If only start-line is specified, then a single line is renumbered and start-line must represent an actual line number.

When a range, start-line TO end-line, is specified, all program lines within the range, inclusive, are renumbered. In this case, start-line and end-line need not be actual line numbers but, if they do not bracket any actual lines, an error message is displayed. The default range FIRST TO LAST is used if no range is specified.

AT LINE-NUMBER: When AT line-number is specified, line numbers are assigned to the program lines within the specified range by assigning line-number to the first program line within the range.

STEP INCREMENT: When STEP increment is specified, line numbers are assigned to the program lines within the specified range by assigning line numbers to program lines within that range by adding the STEP increment to the previously renumbered program line.

All line number references affected by the renumbering are changed to agree with the new line numbers.

The current line setting is unchanged; however, it may have a new line number associated with it.

The RENUMBER command issues an error message for the following:

- Renumbering would cause the overlap of line numbers outside the range specified.
- Renumbering would generate a line number that would exceed the maximum legal line number (9999999).

- Renumbering would resolve previously unresolved line number references.
- Renumbering would cause a record to exceed the maximum record length.

Example 1

```
RENUMBER 180 TO LAST AT 120 STEP 5
```

Renumber the lines from 180 to the end, start with 120 and increment by 5.

Example 2

```
RENUMBER
```

Renumber the entire file currently in the workspace, start with 100 and increment by 10.

RUN COMMAND

The RUN command initiates execution of a program, starting with the lowest numbered statement.

Format 1

```
RUN [file-spec] [([SOURCE][SPREC|LPREC][)]][STEP]
```

Format 2

```
RUN file-spec (OBJECT)
```

Minimum: RU

Where:

file-spec
is a file name.

General Description

If file-spec is specified, the specified program is loaded into the workspace as if the command

```
LOAD file-spec
```

had been executed (see "LOAD Command" on page 297).

Execution is then initiated starting with the lowest numbered line in the workspace.

If file-spec is not specified, the workspace is unchanged.

Format 1 Description

SOURCE OPTION: SOURCE designates execution of a program in the workspace. If the optional keyword SOURCE is not specified, SOURCE is assumed.

SPREC AND LPREC OPTIONS: These options determine the maximum number of digits displayed when the program prints unformatted decimal values:

SPREC 6 digits are displayed. This is the default, as shipped by IBM, which may be overridden by an OPTION statement within the program.

LPREC 12 digits are displayed.

If neither SPREC nor LPREC is specified, the default is used. Check with your system administrator for the default value in force for your organization.

STEP OPTION: The STEP option specifies execution in the step mode. In the step mode, processing halts before executing each statement (including the first) and displays the line number. Thus, you are allowed to interact with the processor between statements.

After processing halts, pressing the ENTER key causes the next statement to be executed and keeps the program in step mode.

There are three ways to exit from STEP mode:

- Response to the step prompt with anything other than the ENTER key, for example, another command. (See "GO Command" on page 289 to continue execution.)
- The program executes a STOP, END, or BREAK statement.
- The program generates an exception that terminates program execution.

Stepping ignores function references. If you are stepping across a statement which refers to a multiline function (DEF/FNEND block), all of the function statements are executed as part of the step. However, if the function contains a line which has been designated as a breakpoint by the BREAK command, or if an attention interrupt with SYSTEM action occurs while executing a multiline function, a break occurs in the function, but stepping mode at the level of the referencing statement is not terminated.

The RUN command initializes the values of all variables, arrays, and ON condition actions. Numeric variables and arrays are set to zero. Character variables and arrays are set to the null string. ON condition actions are set to SYSTEM. All files are closed. Immediate variables are dropped. The STEP option can be used to halt just after this initialization has been done and before the first statement is executed. Immediate LET statements can then be used to initialize variables to other values before continuing execution with the GO command.

Example 1

```
RUN (SPREC) STEP
```

Initiates execution of the program in the workspace in the STEP mode with short precision printing of unformatted numerics (maximum of six decimal digits).

Example 2

```
RUN MYFILE (LPREC)
```

Loads the source code of the program named MYFILE into the workspace and executes it with long precision printing of unformatted numerics (maximum of twelve decimal digits).

Example 3

```
RUN PROGA
```

Loads the program named PROGA in the workspace and executes it.

Example 4

```
RUN MYFILE (SOURCE LPREC)
```

This format is also used to bring a specified program into the workspace and begin execution. It is equivalent to a LOAD command followed by a RUN command with no file-spec.

```
RUN TESTPROG STEP
```

is equivalent to

```
LOAD TESTPROG
RUN STEP
```

Note that the prior contents of your workspace are lost (see "LOAD Command" on page 297).

The program lines are checked for proper syntax as they are brought into the workspace and error messages are displayed.

Format 2 Description

The second form of the RUN command executes a program that has been previously compiled and exists as an object module in your library. The step mode is not available, and SPREC and LPREC are as specified when the program was compiled.

The workspace is cleared, and the named object module is loaded and executed.

The RUN command initializes the values of all variables, arrays, and ON condition actions. The values of all numeric variables and arrays are set to zero. Character variables and arrays are set to the null string. ON condition actions are set to SYSTEM. All files are closed. Immediate variables are dropped. After the object module has been executed, the object module is cleared from the workspace, and the workspace does not have a name.

Example

```
RUN MYFILE (OBJECT)
```

Clears the workspace and executes the object module named MYFILE. The workspace is cleared after executing the object module.

SAVE COMMAND

The SAVE command copies the program in the workspace to a file.

Format

SAVE [file-spec][(REPLACE[)]]

Minimum: SA

Where:

file-spec
is a file name.

Description

The program lines in your workspace are written to a file in line number sequence. The file is determined as follows:

- File-spec, if specified.
- If the workspace has a name associated with it (set by the FETCH, LOAD, INITIALIZE, RENAME, and RUN commands), that name is prompted. You may accept it by responding with the ENTER key, or supply a different name before pressing ENTER.
- If the workspace does not have a name, you are requested to enter a name. You may supply a name or cancel the command by entering a null line.

If the name entered, either as part of the SAVE command or in response to a prompt, is equal to the current name of the workspace, the workspace is saved to the file whether it exists already or not. The keyword REPLACE is unnecessary and is ignored if present.

If a file name other than the one currently naming your workspace is to be saved, you must explicitly state that an existing file is to be replaced. This may be done in two ways; either by the REPLACE keyword or, if REPLACE is not used, a YES response to a message asking if the file should be replaced. If REPLACE is specified and the file does not already exist, an information message is displayed and the file is saved.

Note that the workspace name is not changed.

Example 1

```
SAVE PROGA
```

Writes PROGA from the workspace to your library. A diagnostic is displayed if PROGA already exists in your library.

Example 2

```
SAVE PROGB (REPLACE)
```

Replaces the old version of PROGB in your library with the new version of PROGB.

An informational diagnostic is displayed if no old version of PROGB exists, and the REPLACE parameter is specified.

SET LOG Command

SET LOG COMMAND

The SET LOG command activates or deactivates logging of the BASIC dialog with the terminal. This command may be used only in the CMS environment.

Format

```
SET LOG {[ON][OUT (file-spec)]|OFF}
```

Minimum: SE LOG

Where:

file-spec
is a file name.

Description

SET LOG controls logging of activity at the terminal. "SET LOG ON" activates logging, and "SET LOG OFF" deactivates logging. While logging is active, copies of lines written to the terminal as well as lines read from the terminal are recorded, in the order they occur, to a file.

The logging file is determined by the OUT clause. If an OUT clause is not specified, the default for file-spec is BASLOG. See IBM BASIC Applications Programming: System Services for details of file types.

If none of the parameter keywords are specified, ON is assumed. Thus "SET LOG" is synonymous with "SET LOG ON".

The log records all terminal input/output with the following exceptions:

- Execution of PRINT FIELDS and INPUT FIELDS are not logged.
- HELP mode is not logged. See "HELP Command" on page 291.
- CMS commands are not logged. This includes output from CMS commands and all dialog while in the CMS subset mode. See "SYSTEM Command" on page 317.

Consecutive SET LOG ON commands without an intervening SET LOG OFF are not accepted. The second results in an error message. SET LOG OFF when logging is not "on" also results in an error message.

Each SET LOG ON command with an OUT clause erases the current contents of the specified logging file before initiating logging.

The default file BASLOG is erased by the first SET LOG ON command in a BASIC session, but not by subsequent SET LOG ON commands. Thus, logging to this file may be turned on and off repeatedly without loss of previous output.

SET MSG COMMAND

The SET MSG command controls the content of error messages displayed at the terminal.

Format

SET MSG ([I|W|E|S])[ALL|TEXT]

Minimum: SE MSG

Description

The SET MSG command controls how error message are displayed. A complete message consists of two parts:

1. Codes. This is a 9-character string.

   The first 3 characters are "BAS", for messages produced by the BASIC Processor, or "BLI", for messages produced by the Library.

   The next 5 characters uniquely define the message.

   The final character of the codes indicates the severity level of the message:

   I informative

   W warning

   E error with corrective assumption

   S error with no corrective assumption

2. Text. This is a variable number of characters which explain the reason for the message.

   When ALL is in effect, both the codes and the text are displayed at the terminal.

Example

```
SET MSG(S) ALL
```

is in effect, and an attempt is made at line 20 to access an array element with a subscript that is larger than the corresponding array dimension, the following message is displayed:

```
BLI02001S LINE 20. SUBSCRIPT OUT OF BOUNDS.  PROGRAM EXECUTION
TERMINATED.
```

When TEXT is in effect, only the text portion of the message is displayed at the terminal.

Example

If prior to the above error, the following SET command is entered.

```
SET MSG(S) TEXT
```

the following message is displayed:

```
LINE 20. SUBSCRIPT OUT OF BOUNDS.  PROGRAM EXECUTION TERMINATED.
```

The SET MSG command controls messages displayed at the terminal, not messages written to a listing file as a result of a COMPILE command. Listing messages always include codes.

The default settings for the four message levels are determined as part of the BASIC installation procedure. As the product is distributed by IBM:

- I level messages are TEXT
- W, E, and S level messages are ALL

STORE COMMAND

The STORE command places the program currently in the workspace (the source program along with internal representation of the program) into a file.

Format

STORE [file-spec][(REPLACE[)]]

Minimum: ST

Where:

file-spec
is a file name.

Description

The workspace placed in the file is determined by:

- File-spec, if specified in the STORE command.
- If the workspace currently has a name, that name is displayed as a prompt. You may either accept the displayed name by pressing ENTER or enter a new name and then press ENTER.
- If your workspace does not have a name, you are asked to enter a name. A null response in this case cancels the STORE command.

If the name entered, either as part of the STORE command or in response to a prompt, is equal to the current name of the workspace, the file is stored whether it already exists or not. The keyword REPLACE is unnecessary and is ignored if present.

If the name is not the current name of the workspace, you must explicitly state that an existing file is to be replaced. This may be done either by the REPLACE keyword in the command or, if the keyword is not used, an affirmative response to an IBM BASIC message asking if the file should be replaced.

If REPLACE is specified and the file does not already exist, the file is stored and you are notified by display.

Files created by the STORE command may be reloaded in the workspace with the FETCH command. (See "FETCH Command" on page 286.)

Example 1

```
STORE MYFILE
```

Write the contents of the workspace to MYFILE.

If MYFILE already exists and the name currently associated with the workspace is not MYFILE, the REPLACE parameter must be specified, or a diagnostic is issued asking you if you really want to replace the file.

Example 2

```
STORE MYFILE (REPLACE)
```

Will replace the old version of MYFILE with the new one. When you are ready to continue processing MYFILE, it may be reloaded into the workspace by executing a FETCH command.

SYSTEM COMMAND

The SYSTEM command executes a CMS SUBSET command or enters CMS SUBSET mode. This command may be used only in the CMS environment.

Format

```
SYSTEM ["cms-command"]
```

Minimum: SY

Where:

cms-command
is one of the CMS SUBSET commands.

Description

If "cms command" is included, it must be one of the CMS SUBSET commands in exactly the form used when CMS is in control. (See CMS Command and Macro Reference for a description of these commands.) This allows you to execute a single CMS command as though it were an IBM BASIC command. Be careful when using these commands; some of them can adversely affect your BASIC terminal session. Figure 43 lists the CMS subset commands.

CMS Nucleus-Resident Commands

CP	GENMOD	START
DEBUG	INCLUDE	STATE
ERASE	LOAD	STATEW
FETCH	LOADMOD	

CMS Transient-Area Commands

ACCESS	HELP	RELEASE
ASSGN	LISTFILE	RENAME
COMPARE	MODMAP	SET
DISK	OPTION	SVCTRACE
DLBL	PRINT	SYNONYM
FILEDEF	PUNCH	TAPE
GENDIRT	QUERY	TYPE
GLOBAL	READCARD	

Figure 43. SYSTEM Command—Valid CMS Subset Commands

If the command is rejected by CMS, an error message is displayed. If it causes a CMS screen display (for example, LISTFILE), the BASIC screen is cleared, the CMS screen appears, and the BASIC screen is restored upon completion of the CMS command.

If just the keyword SYSTEM is entered, the BASIC environment is temporarily suspended and the CMS SUBSET environment is entered. The BASIC screen is erased and a CMS screen is displayed with a RUNNING status. While in the CMS SUBSET environment, any CMS commands which run in the transient area may be issued.

To return to BASIC from the CMS SUBSET environment, enter the CMS SUBSET command, RETURN. The BASIC screen as it existed at the time the SYSTEM command was entered is restored.

For further details, see the IBM BASIC Application Programming: System Services manual.

Example 1

```
SYSTEM
```

enters CMS subset mode.

Example 2

```
SYSTEM "LIST * BASIC *"
```

The system takes over the screen and displays all the files of type BASIC.

APPENDIX A. EXCEPTION CODES

The following table specifies the values of the intrinsic function ERR. Less severe exceptions (those whose SYSTEM action does not halt the program) are denoted by underlined codes. A negative value indicates that the exception is not part of the emerging ANS BASIC Standard. The ON statement condition (ON) and EXIT statement condition (EXIT) of each exception code are defined following the error function.

OVERFLOW (1000)		ON	EXIT
1001	Overflow in evaluating numeric constant.	OFLOW	none
1002	Overflow in evaluating numeric expression.	OFLOW	none
1003	Overflow in evaluating numeric intrinsic function.	OFLOW	none
1004	Overflow in evaluating VAL function.	OFLOW	none
1005	Overflow in evaluating numeric array expression.	OFLOW	none
1006	Overflow in numeric datum for (MAT) READ.	OFLOW	none
1007	Overflow in numeric datum for (MAT) INPUT.	OFLOW	none
1008	Overflow in numeric datum for (MAT) FILE INPUT.	OFLOW	none
-1009	Underflow in numeric datum for (MAT) READ.	UFLOW	none
-1010	Underflow in evaluating numeric expression.	UFLOW	none
-1011	Underflow in numeric datum for (MAT) INPUT.	UFLOW	none
-1012	Underflow in evaluating VAL function.	UFLOW	none
-1013	Underflow in numeric datum for (MAT) file INPUT.	UFLOW	none
-1014	Overflow in numeric datum for INPUT FIELDS.	OFLOW	none
-1015	Underflow in numeric datum for INPUT FIELDS.	UFLOW	none
1051	Overflow in evaluating character expression.	SOFLOW	SOFLOW
1052	Overflow in evaluating character array expression.	SOFLOW	SOFLOW
1053	Overflow in character datum for (MAT) READ.	SOFLOW	SOFLOW
1054	Overflow in character datum for (MAT) INPUT.	SOFLOW	SOFLOW
1055	Overflow in character datum for (MAT) WRITE.	SOFLOW	SOFLOW
-1056	Overflow in character datum for INPUT FIELDS.	SOFLOW	none
-1057	Overflow in character datum for PRINT FIELDS.	SOFLOW	none
-1058	Overflow in character datum for (MAT) LINE INPUT.	SOFLOW	SOFLOW
-1059	Overflow in character datum for (MAT) file INPUT.	SOFLOW	SOFLOW

SUBSCRIPT ERRORS (2000)		ON	EXIT
2001	Subscript out of bounds.	ERROR	none
2002	Redimension index less than subscript lower bound.	ERROR	none
-2003	Wrong number of subscripts.	ERROR	none

MATHEMATICAL ERRORS (3000)		ON	EXIT
3001	Division by zero.	ZDIV	none
3002	Negative number raised to nonintegral power.	ERROR	none
3003	Zero raised to negative power.	ZDIV	none
3004	Logarithm of zero or negative number.	ERROR	none
3005	Square root of negative number.	ERROR	none
3006	Zero divisor specified for MOD or REM.	ZDIV	none
3007	Argument of ACOS or ASIN not in range -1 to +1.	ERROR	none
-3008	Absolute value of argument not less than PI*2**50.	ERROR	none
-3009	Argument approaches a singularity.	ERROR	none
-3010	Absolute value of argument not less than 175.366.	ERROR	none

PARAMETER ERRORS (4000)		ON	EXIT
4001	Argument of VAL not a character representation of a numeric constant.	CONV	none
4002	Argument of CHR$ out of range.	ERROR	none
4003	Argument of ORD not a character or mnemonic.	ERROR	none
4004	Invalid index specified for size.	ERROR	none
4005	Index in TAB less than one.	ERROR	none
4006	Negative index for margin setting.	ERROR	none
-4007	Argument of DAT$ or JDY out of range.	ERROR	none
-4008	Argument of POS less than left margin.	ERROR	none
-4101	Common definitions do not match.	ERROR	none
-4102	CHAIN/USE parameter type mismatch.	ERROR	none
-4103	Subroutine parameter type mismatch.	ERROR	none
-4104	Function parameter type mismatch.	ERROR	none

STORAGE EXHAUSTED ERRORS (5000)		ON	EXIT
5001	Size of redimensioned array too large.	ERROR	none
-5003	Insufficient storage for intermediate character result.	ERROR	none
-5004	Insufficient storage for intermediate array result.	ERROR	none

STORAGE EXHAUSTED ERRORS (5000)		ON	EXIT
-5005	Insufficient storage for I/O buffer.	ERROR	none
-5006	Insufficient storage to execute GOSUB or CALL.	ERROR	none
-5007	Insufficient storage to preserve CHAIN data.	ERROR	none
-5008	Insufficient storage for user data area.	ERROR	none
-5009	Insufficient storage for COMMON data area.	ERROR	none

MATRIX ERRORS (6000)		ON	EXIT
6001	Mismatched dimensions in numeric array expression.	ERROR	none
6002	Argument of DET is not a square matrix.	ERROR	none
6003	Reference to DET without argument prior to INV.	ERROR	none
6004	Argument of INV is not a square matrix.	ERROR	none
6005	Argument to IDN does not specify a square matrix.	ERROR	none
-6006	Array argument of AIDX or DIDX is not a one dimensional array.	ERROR	none
-6007	Argument of TRN is not a two dimensional array.	ERROR	none
-6008	Cannot invert a singular matrix.	ERROR	none
-6009	Destination array too small for MAT assignment.	ERROR	none
6101	Mismatched dimensions in character array expression.	ERROR	none

FILE USE ERRORS (7000)		ON	EXIT
7001	Channel number not integer in range 0 to 255.	ERROR	IOERR
7002	Channel number zero not allowed in OPEN, CLOSE, RESET, or SCRATCH.	ERROR	IOERR
7003	Nonzero channel number in OPEN already active.	ERROR	IOERR
7004	Inactive channel number in file statement other than OPEN.	ERROR	IOERR
-7005	ENDPAGE.	ENDPAGE	ENDPAGE
-7006	Device unable to perform requested operation.	ERROR	IOERR
-7007	I/O statement referenced user-defined function which terminated the I/O.	ERROR	IOERR
7201	File pointer cannot be reset as specified.	ERROR	IOERR
7202	RECORD clause not allowed for nonrelative file. file.	ERROR	IOERR
7203	KEY clause not allowed for nonkeyed file.	ERROR	IOERR
-7204	File pointer cannot be reset to KEY as specified.	ERROR	NOKEY
-7205	File pointer cannot be reset to RECORD as specified.	ERROR	NOREC
7301	Scratch of file not permitted.	ERROR	IOERR

FILE USE ERRORS (7000)		ON	EXIT
7302	Output not allowed to file opened for INPUT.	ERROR	IOERR
7303	Input not allowed from file opened for OUTPUT.	ERROR	IOERR
7304	Line input not allowed from nondisplay format file.	ERROR	IOERR
7305	Record length specified exceeds current record length.	ERROR	IOERR
7306	Attempt to rewrite keyed record with different key.	ERROR	IOERR
7307	Attempt to rewrite nonexistent relative record.	ERROR	NOREC
7308	Attempt to write existing relative record.	ERROR	DUPREC
-7309	Attempt to write existing record.	ERROR	IOERR
-7310	Invalid operation for type of file.	ERROR	IOERR
-7311	GET or PUT not allowed for display file.	ERROR	IOERR
-7312	FORM statement not allowed for internal file.	ERROR	IOERR
-7313	File not opened in OUTIN mode.	ERROR	IOERR
-7314	FORM statement required for native file.	ERROR	IOERR
-7315	Record truncated on READ.	ERROR	IOERR
-7316	Position end not allowed for file opened for INPUT.	ERROR	IOERR
-7317	Operation not allowed for native format file.	ERROR	IOERR
-7318	Open OUTPUT or write for file that is read only.	ERROR	IOERR
-7319	Character overflow on PRINT FORM to IMAGE/SPEC.	SOFLOW	SOFLOW
-7320	File does not exist.	ERROR	IOERR
-7321	OUTIN only allowed for native or internal files with record organization.	ERROR	IOERR
-7322	End media or extents.	ERROR	EOF
-7323	Invalid file organization.	ERROR	IOERR
-7324	Records must be fixed length for relative file.	ERROR	IOERR
-7325	Invalid reset or position end on relative file.	ERROR	IOERR
-7326	Invalid reset or position end on keyed file.	ERROR	IOERR
-7327	Device 3800 expected but not found.	ERROR	IOERR
-7328	Display file must have sequential organization.	ERROR	IOERR
-7329	Internal file must have sequential or stream organization.	ERROR	IOERR
-7330	Stream organization not allowed for native file.	ERROR	IOERR
-7331	Margin error: Not enough room for a print zone.	ERROR	IOERR
-7332	Margin error: Bottom margin is less than top margin.	ERROR	IOERR
-7333	Margin statement only allowed for display files.	ERROR	IOERR
-7335	Record length exceeded on OUTPUT to file.	CONV	CONV

FILE USE ERRORS (7000)		ON	EXIT
-7336	Attempt to write a relative record without a REC clause.	ERROR	IOERR
-7337	Invalid form specification for PRINT statement.	ERROR	IOERR
-7338	FONT may be specified for 3800 DEVICE only.	ERROR	IOERR
-7339	Record length specified exceeds maximum record length of file.	ERROR	IOERR
-7340	Filename syntax error.	ERROR	IOERR
-7341	DEVICE 3800 only valid for display files.	ERROR	IOERR
-7342	Illegal file operation for channel number zero.	ERROR	IOERR
-7343	Invalid record length specified.	ERROR	IOERR
-7344	Invalid filetype specified.	ERROR	IOERR
-7345	Invalid filemode specified.	ERROR	IOERR
-7346	READ required before REREAD.	ERROR	IOERR
-7347	Invalid margin value.	ERROR	IOERR
-7348	READ required before REWRITE.	ERROR	IOERR
-7350	Invalid INPUT operation with file pointer set at end by WRITE.	ERROR	IOERR
-7351	Right margin error.	ERROR	IOERR
-7352	Print zone size too small.	ERROR	IOERR
-7353	Font value must be from 1 through 4.	ERROR	IOERR
-7354	Invalid password.	ERROR	IEORR
-7355	Attempt to read nonexistent keyed record.	ERROR	NOKEY
-7356	Attempt to read nonexistent relative record.	ERROR	NOREC
-7357	Attempt to delete nonexistent keyed record.	ERROR	NOKEY
-7358	Attempt to delete nonexistent relative record.	ERROR	NOREC
-7359	Attempt to rewrite nonexistent keyed record.	ERROR	NOKEY
-7360	Attempt to write existing keyed record.	ERROR	DUPKEY
-7401	PF KEY ignored.	SKEY	none

INPUT/OUTPUT ERRORS (8000)		ON	EXIT
8001	(MAT) READ beyond end of data.	ERROR	IOERR
8002	Too few data items in input reply.	ERROR	IOERR
8003	Too many data items in input reply.	ERROR	IOERR
8011	End-of-file encountered on input.	ERROR	EOF
8012	Too few data items in record.	CONV	CONV
8013	Too many data items in record.	CONV	CONV

INPUT/OUTPUT ERRORS (8000)		ON	EXIT
-8016	Attempt to position beyond end of record.	CONV	CONV
-8017	READ with no data statements declared in program unit.	ERROR	IEORR
8101	Nonnumeric datum for (MAT) READ of numeric item.	CONV	CONV
8103	Noncharacter datum for (MAT) INPUT of number.	CONV	CONV
8104	Nonstring datum for (MAT) INPUT of string from file.	CONV	CONV
-8105	Nonnumeric datum for (MAT) OUTPUT for FORM numeric specification.	CONV	CONV
-8106	Numeric data for C or V format item.	CONV	CONV
-8107	Nonnumeric datum for file input of numeric item.	CONV	CONV
-8108	Nonnumeric datum for INPUT FIELDS of numeric item.	CONV	none
-8109	Syntactically incorrect data for file input.	ERROR	IOERR
-8110	Syntactically incorrect data for (MAT) read.	ERROR	IOERR
-8111	Noncharacter input list item specified for LINE INPUT.	CONV	CONV
8201	Invalid format string.	ERROR	IOERR
8202	Data conversion specification in FORM or IMAGE.	ERROR	IOERR
-8203	Missing comma in FORM.	ERROR	IOERR
-8204	IMAGE allowed only for display files.	ERROR	IOERR
-8205	IMAGE specification exceeds right margin.	CONV	CONV
-8206	Replication count must be greater than zero.	ERROR	IOERR
-8208	Replication count must be integer type.	ERROR	IEORR
-8209	Replication count exceeds maximum allowed.	ERROR	IOERR
-8210	Expecting comma separator between input items.	ERROR	IOERR
-8211	No closing quote found on quoted string.	ERROR	IOERR
-8212	Trailing quote with no matching leading quote.	ERROR	IOERR
-8213	Invalid specification for replication factor.	ERROR	IOERR
-8214	Missing closing quote for character string in form.	ERROR	IOERR
-8215	Form NC conversion width limited to 255.	ERROR	IOERR
-8301	IMAGE output item specified without an IMAGE output specification.	ERROR	IOERR
-8302	Floating representation not allowed for N data FORM specification.	CONV	CONV
-8304	FORM B specification must have a width of 2, 4, or 8.	ERROR	IOERR
-8305	Syntax error in FORM—invalid width specification.	ERROR	IOERR
-8306	PIC may not be used with READ.	ERROR	IOERR

INPUT/OUTPUT ERRORS (8000)		ON	EXIT
-8307	Numeric conversion syntax error.	ERROR	IOERR
-8308	Syntax error in FORM—missing width specification.	ERROR	IOERR
-8309	Syntax error in FORM—invalid decimal specification.	ERROR	IOERR
-8310	FORM PIC specification must be enclosed in parentheses.	ERROR	IOERR
-8311	Replication count not allowed for input fields reply.	ERROR	IOERR
-8312	Input field definition overlaps an existing attribute character.	ERROR	IOERR
-8313	INPUT (PRINT) fields: Invalid syntax for row number.	ERROR	none
-8314	INPUT (PRINT) fields: Row number must be between 1 and maximum allowed.	ERROR	none
-8315	INPUT (PRINT) fields: Invalid syntax for column number.	ERROR	none
-8316	INPUT (PRINT) fields: Column number must be between 1 and maximum line length.	ERROR	none
-8317	INPUT (PRINT) fields: Invalid syntax for field length.	ERROR	none
-8318	INPUT (PRINT) fields: Field length must be between 1 and 156.	ERROR	none
-8319	INPUT (PRINT) fields: Attribute character must be H,I, or N.	ERROR	none
-8321	PIC syntax error.	ERROR	IOERR
-8324	Form scale specification is greater than width specification.	ERROR	IOERR
-8326	Numeric value will not fit in form specification.	CONV	CONV
-8327	Syntactically incorrect data for INPUT FIELDS.	ERROR	IOERR
-8328	Numeric value will not fit in IMAGE specification. Replaced with asterisks.	CONV	CONV

DEVICE ERRORS (9000)

(Not used)

CONTROL ERRORS(10000)		ON	EXIT
10001	Index out of range, no ELSE in ON#GOTO or ON GOSUB.	ERROR	none
10002	Return without corresponding GOSUB or ON GOSUB.	ERROR	none
10003	No CASE block selected and no CASE ELSE.	ERROR	none
10004	Attempt to chain to unavailable program.	ERROR	none
-10005	Recursive function reference.	ERROR	none

CONTROL ERRORS(10000)		ON	EXIT
-10006	Attempt to execute a previously diagnosed erroneous statement.	ERROR	none
-10008	Retry or continue with no active exception.	ERROR	none
-10009	Reference to undefined line number or label.	ERROR	none
-10010	Invalid exception transfer to nonactive user function.	ERROR	none
-10011	Invalid transfer of control.	ERROR	none
-10012	Unable to load CHAINed program unit.	ERROR	none
-10013	Unable to load CALLed program unit.	ERROR	none
-10014	Unable to perform chain.	ERROR	none

SYSTEM ERRORS (11000)		ON	EXIT
-11001	System error returned on INPUT.	ERROR	IOERR
-11002	System error returned on CALL SYSTEM.	ERROR	none
-11003	System error returned on RESET.	ERROR	IOERR
-11004	System error returned on READ.	ERROR	IOERR
-11005	System error returned on PRINT.	ERROR	IOERR
-11006	System error returned on DELETE.	ERROR	IOERR
-11007	System error returned on SCRATCH.	ERROR	IOERR
-11008	System error returned on CLOSE.	ERROR	IOERR
-11009	System error returned on REWRITE.	ERROR	IOERR
-11010	System error returned on WRITE.	ERROR	IOERR
-11011	System error returned on OPEN.	ERROR	IOERR
-11012	System error returned on INPUT fields.	ERROR	IOERR
-11013	System error returned on PRINT fields.	ERROR	IOERR
-11014	System error returned on PAUSE.	ERROR	IOERR

SPECIAL (12000)		ON	EXIT
-12001	Program suspended due to attention.	ATTN	none
-12002	Unrecognized exception generated by CAUSE statement.	ERROR	none

APPENDIX B. CHARACTER SET COLLATING SEQUENCES

This appendix lists the ASCII and EBCDIC collating sequences.

ASCII CHARACTER SET AND COLLATING SEQUENCE

ASCII is the American National Standard Code for Information Interchange.

In an IBM BASIC program this collating sequence and character set is specified through the OPTION COLLATE STANDARD statement.

ASCII SEQUENCE AND CHARACTER SET

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
0	0/0		NUL	Null
1	0/1		SOH	Start of heading
2	0/2		STX	Start of text
3	0/3		ETX	End of text
4	0/4		EOT	End of transmission
5	0/5		ENQ	Enquiry
6	0/6		ACK	Acknowledge
7	0/7		BEL	Bell
8	0/8		BS	Backspace
9	0/9		HT	Horizontal tab
10	0/10		LF	Line feed
11	0/11		VT	Vertical tab
12	0/12		FF	Form feed
13	0/13		CR	Carriage Return
14	0/14		SO	Shift out
15	0/15		SI	Shift in
16	1/0		DLE	Data link escape
17	1/1		DC1	Device control 1
18	1/2		DC2	Device control 2
19	1/3		DC3	Device control 3
20	1/4		DC4	Device control 4
21	1/5		NAK	Negative acknowledge
22	1/6		SYN	Synchronous idle
23	1/7		ETB	End of trans. block
24	1/8		CAN	Cancel
25	1/9		EM	End of medium
26	1/10		SUB	Substitute
27	1/11		ESC	Escape
28	1/12		FS	File separator
29	1/13		GS	Group separator
30	1/14		RS	Record separator
31	1/15		US	Unit separator
32	2/0		SP	Space
33	2/1	!		Exclamation mark
34	2/2	"		Quotation mark
35	2/3	#		Number sign
36	2/4	$		Dollar sign
37	2/5	%		Percent sign
38	2/6	&		Ampersand
39	2/7	'		Apostrophe
40	2/8	(		Left parenthesis
41	2/9	)		Right parenthesis
42	2/10	*		Asterisk
43	2/11	+		Plus
44	2/12	,		Comma
45	2/13	-		Minus sign
46	2/14	.		Full stop
47	2/15	/		Solidus
48	3/0	0		Zero
49	3/1	1		One

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
50	3/2	2		Two
51	3/3	3		Three
52	3/4	4		Four
53	3/5	5		Five
54	3/6	6		Six
55	3/7	7		Seven
56	3/8	8		Eight
57	3/9	9		Nine
58	3/10	:		Colon
59	3/11	;		Semicolon
60	3/12	<		Less than sign
61	3/13	=		Equals sign
62	3/14	>		Greater than sign
63	3/15	?		Question mark
64	4/0	@		Commercial at
65	4/1	A		Uppercase A
66	4/2	B		Uppercase B
67	4/3	C		Uppercase C
68	4/4	D		Uppercase D
69	4/5	E		Uppercase E
70	4/6	F		Uppercase F
71	4/7	G		Uppercase G
72	4/8	H		Uppercase H
73	4/9	I		Uppercase I
74	4/10	J		Uppercase J
75	4/11	K		Uppercase K
76	4/12	L		Uppercase L
77	4/13	M		Uppercase M
78	4/14	N		Uppercase N
79	4/15	O		Uppercase O
80	5/0	P		Uppercase P
81	5/1	Q		Uppercase Q
82	5/2	R		Uppercase R
83	5/3	S		Uppercase S
84	5/4	T		Uppercase T
85	5/5	U		Uppercase U
86	5/6	V		Uppercase V
87	5/7	W		Uppercase W
88	5/8	X		Uppercase X
89	5/9	Y		Uppercase Y
90	5/10	Z		Uppercase Z
91	5/11	[		Left bracket
92	5/12	\		Reverse solidus
93	5/13	]		Right bracket
94	5/14	¬ or ^		Logical NOT (also circumflex accent)
95	5/15	_	UND	Underline
96	6/0	`	GRA	Grave accent
97	6/1	a	LCA	Lowercase a
98	6/2	b	LCB	Lowercase b
99	6/3	c	LCC	Lowercase c
100	6/4	d	LCD	Lowercase d

ASCII SEQUENCE AND CHARACTER SET

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
101	6/5	e	LCE	Lowercase e
102	6/6	f	LCF	Lowercase f
103	6/7	g	LCG	Lowercase g
104	6/8	h	LCH	Lowercase h
105	6/9	i	LCI	Lowercase i
106	6/10	j	LCJ	Lowercase j
107	6/11	k	LCK	Lowercase k
108	6/12	l	LCL	Lowercase l
109	6/13	m	LCM	Lowercase m
110	6/14	n	LCN	Lowercase n
111	6/15	o	LCO	Lowercase o
112	7/0	p	LCP	Lowercase p
113	7/1	q	LCQ	Lowercase q
114	7/2	r	LCR	Lowercase r
115	7/3	s	LCS	Lowercase s
116	7/4	t	LCT	Lowercase t
117	7/5	u	LCU	Lowercase u
118	7/6	v	LCV	Lowercase v
119	7/7	w	LCW	Lowercase w
120	7/8	x	LCX	Lowercase x
121	7/9	y	LCY	Lowercase y
122	7/10	z	LCZ	Lowercase z
123	7/11	{	LBR	Left brace
124	7/12	|	VLN	Vertical line
125	7/13	}	RBR	Right brace
126	7/14	~	TIL	Tilde
127	7/15		DEL	Delete

EBCDIC CHARACTER SET AND COLLATING SEQUENCE

EBCDIC is the Extended Binary Coded Decimal Interchange Code.

In an IBM BASIC program this collating sequence and character set is specified through the OPTION COLLATE NATIVE statement.

EBCDIC SEQUENCE AND CHARACTER SET

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
0	0/0		NUL	Null
1	0/1		SOH	Start of heading
2	0/2		STX	Start of text
3	0/3		ETX	End of text
4	0/4		PF	Punch off
5	0/5		HT	Horizontal tab
6	0/6		LC	Lowercase
7	0/7		DEL	Delete
8	0/8		GE	
9	0/9		RLF	
10	0/10		SMM	Start of manual message
11	0/11		VT	Vertical tab
12	0/12		FF	Form feed
13	0/13		CR	Carriage return
14	0/14		SO	Shift out
15	0/15		SI	Shift in
16	1/0		DLE	Data link escape
17	1/1		DC1	Device control 1
18	1/2		DC2	Device control 2
19	1/3		TM	Tape mark
20	1/4		RES	Restore
21	1/5		NL	New line
22	1/6		BS	Backspace
23	1/7		IL	Idle
24	1/8		CAN	Cancel
25	1/9		EM	End of medium
26	1/10		CC	Cursor control
27	1/11		CU1	Customer use 1
28	1/12		IFS	Interchange file separator
29	1/13		IGS	Interchange group separator
30	1/14		IRS	Interchange record separator
31	1/15		IUS	Interchange unit separator
32	2/0		DS	Digit select
33	2/1		SOS	Start of significance
34	2/2		FS	File separator
36	2/4		BYP	Bypass
37	2/5		LF	Line feed
38	2/6		ETB	End of trans. block
39	2/7		ECS	Escape
42	2/10		SM	Set mode
43	2/11		CU2	Customer use 2
45	2/13		ENQ	Enquiry
46	2/14		ACK	Acknowledge
47	2/15		BEL	Bell
50	3/2		SYN	Synchronous idle

EBCDIC SEQUENCE AND CHARACTER SET

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
52	3/4		PN	Punch on
53	3/5		RS	Record separator
54	3/6		UC	Uppercase
55	3/7		EOT	End of transmission
59	3/11		CU3	Customer use 3
60	3/12		DC4	Device control 4
61	3/13		NAK	Negative acknowledge
63	3/15		SUB	Substitute
64	4/0		SP	Space
74	4/10	¢		Cent sign
75	4/11	.		Period, decimal point
76	4/12	<		Less than sign
77	4/13	(		Left parenthesis
78	4/14	+		Plus sign
80	5/0	&		Ampersand
90	5/10	!		Exclamation mark
91	5/11	$		Dollar sign
92	5/12	*		Asterisk
93	5/13	)		Right parenthesis
94	5/14	;		Semicolon
95	5/15	¬ or ^		Logical NOT (also circumflex accent)
96	6/0	-		Minus sign
97	6/1	/		Slash
107	6/11	,		Comma
108	6/12	%		Percent sign
109	6/13	_	UND	Underscore
110	6/14	>		Greater than sign
111	6/15	?		Question mark
121	7/9	`	GRA	Grave accent
122	7/10	:		Colon
123	7/11	#		Number sign
124	7/12	@		Commercial at
125	7/13	'		Apostrophe
126	7/14	=		Equal sign
127	7/15	"		Quotation mark
129	8/1	a	LCA	Lowercase a
130	8/2	b	LCB	Lowercase b
131	8/3	c	LCC	Lowercase c
132	8/4	d	LCD	Lowercase d
133	8/5	e	LCE	Lowercase e
134	8/6	f	LCF	Lowercase f
135	8/7	g	LCG	Lowercase g
136	8/8	h	LCH	Lowercase h
137	8/9	i	LCI	Lowercase i
145	9/1	j	LCJ	Lowercase j
146	9/2	k	LCK	Lowercase k
147	9/3	l	LCL	Lowercase l

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
148	9/4	m	LCM	Lowercase m
149	9/5	n	LCN	Lowercase n
150	9/6	o	LCO	Lowercase o
151	9/7	p	LCP	Lowercase p
152	9/8	q	LCQ	Lowercase q
153	9/9	r	LCR	Lowercase r
161	10/1	~		Tilde
162	10/2	s	LCS	Lowercase s
163	10/3	t	LCT	Lowercase t
164	10/4	u	LCU	Lowercase u
165	10/5	v	LCV	Lowercase v
166	10/6	w	LCW	Lowercase w
167	10/7	x	LCX	Lowercase x
168	10/8	y	LCY	Lowercase y
169	10/9	z	LCZ	Lowercase z
192	12/0	{	LBR	Left Brace
193	12/1	A		Uppercase A
194	12/2	B		Uppercase B
195	12/3	C		Uppercase C
196	12/4	D		Uppercase D
197	12/5	E		Uppercase E
198	12/6	F		Uppercase F
199	12/7	G		Uppercase G
200	12/8	H		Uppercase H
201	12/9	I		Uppercase I
208	13/0	}	RBR	Right brace
209	13/1	J		Uppercase J
210	13/2	K		Uppercase K
211	13/3	L		Uppercase L
212	13/4	M		Uppercase M
213	13/5	N		Uppercase N
214	13/6	O		Uppercase O
215	13/7	P		Uppercase P
216	13/8	Q		Uppercase Q
217	13/9	R		Uppercase R
224	14/0	\		Reverse solidus
226	14/2	S		Uppercase S
227	14/3	T		Uppercase T
228	14/4	U		Uppercase U
229	14/5	V		Uppercase V
230	14/6	W		Uppercase W
231	14/7	X		Uppercase X
232	14/8	Y		Uppercase Y
233	14/9	Z		Uppercase Z
240	15/0	0		Zero
241	15/1	1		One
242	15/2	2		Two
243	15/3	3		Three
244	15/4	4		Four
245	15/5	5		Five
246	15/6	6		Six
247	15/7	7		Seven
248	15/8	8		Eight
249	15/9	9		Nine

APPENDIX C. MIGRATION FROM VS BASIC

Migration from VS BASIC to IBM BASIC includes considerations of language, intrinsic functions, file structures, and arithmetic, as explained in the following paragraphs.

LANGUAGE

Some of the same statements in IBM BASIC and VS BASIC have different clauses and options. The syntax of these statements must be changed to conform to IBM BASIC. Among them are:

CHAIN, CLOSE, DELETE, DIM (for character variables), END, FORM, FOR/NEXT, GET, GOSUB (computed), GOTO (computed), ON, OPEN, PRINT USING, PUT, READ (FILE), REM, RESET, RETURN, REWRITE (FILE), STOP, WRITE

The INPUT FROM statement is replaced by the INPUT fileref statement, and the PRINT TO statement is replaced by the PRINT fileref statement.

If logical, relational, or concatenation operators are used, they must be changed to conform to IBM BASIC usage.

The default for the lower bound of a subscript in IBM BASIC is 0; in VS BASIC it is 1. (In an IBM BASIC program setting OPTION BASE 1 gives the same effect as in VS BASIC.)

IBM BASIC uses variable-length strings; VS BASIC uses fixed-length strings.

INTRINSIC FUNCTIONS

Some of the names of the intrinsic functions are different, although they perform the same functions.

FILE STRUCTURES

VSAM files created by IBM BASIC programs can be processed by VS BASIC programs, and vice versa.

IBM BASIC and VS BASIC files created through the native system access method are not compatible.

ARITHMETIC

The magnitude and precision of numeric data differ between IBM BASIC and VS BASIC. Increased accuracy of IBM BASIC may result in different solutions to mathematical calculations.

IBM BASIC rounding and truncation rules differ from VS BASIC rules.

Not all the VS BASIC predefined constants exist in IBM BASIC.

Exponentiation operators may be different. (VS BASIC uses the up-arrow exponentiation symbol; IBM BASIC uses the circumflex (or NOT sign) exponentiation symbol.)

VS BASIC DATA SET MIGRATION

IBM BASIC programs can read both stream-oriented and record-oriented VS BASIC files. If the specific data layout is known, IBM BASIC can handle VS BASIC stream-oriented files as IBM BASIC native format files, using FORM specifications.

With some restrictions, IBM BASIC programs can read VS BASIC record-oriented files either as IBM BASIC display format files or as IBM BASIC native format files.

GLOSSARY

The terms in this glossary are defined in accordance with their meanings in BASIC. These terms may or may not have the same meanings in other languages.

Definitions from the Draft Proposed American National Standard for BASIC dated February 15, 1982, are preceded by (B).

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from American National Standard Vocabulary for Information Processing, ANSI X3.12-1970 (copyright 1970 by American National Standards Institute, Inc.), which was prepared by the subcommittee on Terminology and Glossary, X3.5.

American National Standard definitions are preceded by an asterisk (*).

Alphabetic Character. Any of the 26 letters (A through Z) of the English alphabet, or any of the alphabet extenders (#, @, and $).

Argument. An item appearing in parentheses in a function reference or a subprogram CALL statement. The item represents a value (or array of values) that the function or subprogram is to act upon. An argument can be a numeric or character expression, and, in the CALL statement, an empty array declarator.

Arithmetic constant. A constant with a numeric value. The forms of arithmetic constants permitted in IBM BASIC are integer and decimal.

Arithmetic data item. Data having a numeric value

Arithmetic expression. An arithmetic constant, a simple arithmetic variable, a reference to an element of an arithmetic array, an arithmetic-valued function reference, or a sequence of the above appropriately separated by numeric operators and parentheses.

Arithmetic variable. The name of an arithmetic data item whose value is assigned and/or changed during program execution. The name consists of 1 to 40 characters. The first character must be an alphabetic character, and the remaining characters may be alphabetic characters, digits, or the underline character. The last character can also be a number sign (signifying a "decimal" variable) or percent sign (signifying an "integer" variable).

Array. (1) * An arrangement of elements in one or more dimensions. (2) In BASIC, a named list or table of data items, all of which are the same type, arithmetic or character, and all of which have the same maximum length. IBM BASIC allows up to seven dimensional arrays.

Array declaration. The process of naming an array and assigning dimensions to it either explicitly (by the DIM or COMMON statement) or implicitly through usage.

Array declarator. An array name followed by a list of dimensions enclosed in parentheses. Used in DIM and COMMON statements to establish the dimensions of arrays. Empty array declarators (array declarators with no dimensions, just parentheses and, possibly, commas) are used in CALL and SUB statements to pass array arguments and declare array parameters, respectively.

Array element. See Array member.

Array expression. An arithmetic expression or a character expression representing an array of values rather than a single value. It may be used only in an array assignment statement (MAT statement).

Array member. A single data item in an array; its position is indicated by a subscripted array reference.

Array name. The name of an array, which follows the rules for formation of a variable name. See Variable name.

*** ASCII.** American National Code for Information Interchange. The standard code using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

Assignment. The process of giving values to variables; for example, via LET statements, READ statements, INPUT statements, etc.

Assignment statement. A statement that moves data from one variable to another, internally.

Assignment symbol. The symbol =, which is used in an assignment statement to give a value to one or more variables.

(B) BASIC. A term applied as a name to members of a special class of languages which possess similar syntaxes and semantic meanings; acronym for Beginner's All-purpose Symbolic Instruction Code.

(B) Batch mode. The processing of programs in an environment where no provision is made for user interaction.

Binary operator. A numeric operator having two terms. The binary operators that can be used in absolute or relocatable expressions and arithmetic expressions are: addition (+), subtraction (-), multiplication (*), and division (/).

*** Branch.** In the execution of a computer program, to select one from a number of alternative sets of instructions.

Built-in function. See Intrinsic function.

Character constant. A constant with a character value. It is usually enclosed by quotation marks, but character constants in DATA statements and INPUT replies may not have enclosing quotation marks.

Character data. Data having a character value, as opposed to a numeric value.

Character expression. A character constant, a simple character variable, a reference to a character array element, a character-valued function reference, a substring of a character variable or array element, or a sequence of the above appropriately separated by concatenation operators and parentheses.

Character string. (1) * A string consisting solely of characters. (2) A connected sequence of characters.

Character variable. A character data item whose value is assigned and/or changed during program execution.

Character variable name. The name of a character variable, consisting of 1 to 40 characters, the first of which must be alphabetic and the last of which must be the dollar sign character ($). Intermediate characters may be alphabetic characters, digits, or the underline character.

CMS. Conversational Monitor System.

Collate native. To sequence data in the EBCDIC mode. (See Appendix C.)

Collate standard. To sequence data in the ASCII mode. (See Appendix C.)

Comment. A remark or note included in the body of a program by the programmer. It has no effect on the execution of the program; it merely documents the program. Comments are written as a string of characters and may appear as a part of a REM statement, or without the REM keyword, by using the exclamation mark (!). May be included on all BASIC statements except the DATA and IMAGE statements.

Concatenation. The operation that joins two strings in the order specified, thus forming one string whose length is equal to the sum of the lengths of the two strings. It is specified by the operator &.

Constant. A value that never changes. IBM BASIC has two types of constants: arithmetic and character.

Control specification. One of the specifications X, POS, SKIP, or PAGE used in the FORM statement to specify formatting of records in record-oriented files, or to control print line formatting at a terminal.

Control statement. See FILE I/O.

CP. Control Program.

CRT. Cathode Ray Tube.

Current line. The line at which operations are being performed, or at which operations on a group of lines begin.

Data file. See File.

Data form specification. (1) One of the specifications B, C, NC, ZD, PD, S, L, or PIC, used in the FORM statement to specify formatting of character and numeric values in record-oriented files. (2) One of the specifications C or PIC, used in the FORM statement to format character and numeric values on a printed line. (3) One of the specifications used in IMAGE to specify formatting of character and numeric values on a printed line.

Data item. A single unit of data; that is, a constant, a variable, an array element, or a function reference.

Data table. The values contained in the DATA statements of your program. DATA statements are processed in statement number sequence (lowest to highest). The values in each DATA statement are collected and placed in a single table in order of their appearance (left to right).

Data table pointer. An indicator that moves sequentially through the data table, pointing to each value as it is assigned to a corresponding variable in a READ statement. Initially, the indicator refers to the first item in the table. It can be repositioned to the beginning of the table at any time by the RESTORE statement.

Declaration. See Explicit declaration and Implicit declaration.

Declarative statement. A statement that explicitly types identifiers, or dimensions arrays. Also specifies variables and arrays placed in a common

region of storage that can be shared by the main program and/or several subprograms.

*** Delimiter.** A flag that groups or separates words or values in a line of input.

Digits. Any of the numerals 0 through 9.

Dimension specification. The specification of the size of an array and the arrangement of its members into from one to seven dimensions.

Direct access. The facility to obtain data from a storage device, or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

Display format file. A sequentially organized file designed to be output in human readable form.

EBCDIC. External Binary Coded Decimal Interchange Code. A coded character set consisting of 8-bit coded characters.

EBCDIC collating sequence. The ordering of character data items according to the Extended Binary Coded Decimal Interchange Code.

(B) End-of-line. The character(s) or indicator which indentifies the termination of a line. Lines of three kinds may be identified in BASIC: program lines, print lines, and input-reply lines. End-of-lines may vary between the three cases and may also vary depending upon context. Thus, for example, the end-of-line in an input-reply may vary on a given system depending on the source for input being used in interactive or batch mode.

Typical examples of end-of-line are carriage-return, carriage-return line-feed, and end of record (such as end of card).

(B) Error. A flaw in the syntax of a program which causes the program to be incorrect.

*** Error message.** An indication that an error has been detected.

(B) Exception. A circumstance arising in the course of executing a program when an implementation recognizes that the semantic rules of the BASIC standard cannot be followed or that some resource constraint is about to be exceeded. Certain exceptions may be handled by automatic recovery procedures specified in the BASIC standard. These and other exceptions may also be handled by recovery procedures specified in the program. If no recovery procedure is given or if restrictions imposed by the hardware or operating environment make it impossible to follow the given procedure, then the exception shall be handled by terminating the program.

Exception statement. A statement provided to allow processing of one or more possible error conditions identified during program execution.

Executable statement. A program statement that causes an action to be performed by the computer.

*** Execution.** The process of carrying out an instruction by the computer.

Execution error. An error discovered during execution of an IBM BASIC program (for example, dividing by zero, branching to a nonexisting statement number, etc.)

Explicit declaration. The use of a DIM or COMMON statement to specify the number of members in an array, the number of dimensions in an array, or the length of a character variable. The use of DECIMAL and INTEGER statements to specify the numeric type of variables, arrays, and functions.

Exponent (of floating-point format number). An integer constant specifying the power of ten by which the base (mantissa) of the decimal floating-point number is to be multiplied.

Exponentiation. Raising a value to a power.

Expression. A notation, within a program, that represents a value: a constant or a reference appearing alone, or combinations of constants and/or references with operators. Four forms of expressions are defined in IBM BASIC: numeric, character, relational, and logical.

*** File.** A set of related records treated as a unit, for example, in stock control, a file could consist of a set of invoices.

File I/O statement. An instruction that transmits data to, or receives data from, a set (file) of similar items that have been grouped together. File I/O statements must include a file reference number to indicate the specific device containing the required file.

Filename. The name of a file.

Fixed-point. A mathematical notation (as in a decimal system) in which the point separating whole numbers and fractions is fixed. See Floating-point.

Fixed-point constant. An arithmetic constant consisting of one or more digits and a decimal point, and optionally preceded by a sign.

Fixed-point format. The form used to express a fixed-point constant.

Floating-point. Involving or being a mathematical notation in which a quantity is denoted by one number multiplied by a power of the number base. See Fixed-point.

Floating point constant. An arithmetic constant consisting of an integer or fixed-point constant (the mantissa) followed by the letter E, followed by an optionally signed one- or two-digit integer constant (the exponent).

Floating-point format. The form used to express a floating-point constant.

Function. A named expression or block of statements that computes a single value. See also Intrinsic function and User-written function.

Function reference. The appearance of an Intrinsic function name or a user-written function name in an expression.

Hard copy. A printed copy of machine output in a visually readable form; for example, printed reports, listings, documents, and summaries.

(B) Identifier. A character string used to name a variable, an array, a function, a picture-definition, subprogram, or a program.

Implicit declaration. (1) The specification of the number of members in an array or the number of dimensions in an array, either by a reference to a member of an array or by context (without the array being explicitly specified in a DIM statement). (2) The specification of the length of a character variable by context (without the variable being explicitly defined in a DIM statement).

Input. The transfer of data from an external medium to internal storage.

Input list. A list of variables to which values are assigned from input data; the list can be made up of scalar variables, array member references, array references, and array references with redimensioning.

Input/output. The transfer of data between an external medium (that is, the terminal typewriter or a file) and a workspace.

Input/output statement. An IBM BASIC statement whose primary function is to transmit data to or from an executing program.

(B) Interactive mode. The processing of programs in an environment which permits you to respond directly to the actions of individual programs and to control the initiation and termination of these programs.

*** Integer.** One of the numbers +1, -1, +2, -2 ... Synonymous with integral number.

Integer constant. An arithmetic constant containing one or more digits, optionally preceded by a sign.

Integer format. The form used to express an integer constant.

Internal format file. A file created by IBM BASIC WRITE, or PUT statements containing self-identifying, sequentially organized data.

Internal text file. A sequential file made up of both binary and EBCDIC data records in a format unique to the IBM BASIC interpreter.

*** Interrupt.** To stop a process in such a way that it can be resumed.

Intrinsic function. A function supplied by IBM BASIC (for example, SIN, COS, SQR, etc.) See Function.

Involution. See Exponentiation.

*** Key.** One or more characters, within a set of data, that contains information about the set, including its identification.

Keyed file. A record-oriented file whose records are stored and accessed according to key values embedded in the records.

(B) Keyword. A character string, usually with the spelling of a commonly used or mnemonic word, which provides a distinctive identification of a statement or a component of a statement of a programming language. (See Reserved word.)

Letter. Any of the uppercase or lowercase characters A through Z, or a through z.

Line. An ordered sequence of characters which terminates with an end-of-line.

Listing files. Files with records that contain only EBCDIC characters; such files can be listed on a printer.

Literal. A symbol or quantity in a source program that is itself data, rather than a reference to data. See Constant.

Logical expression. An expression which uses logical operators to compare two relational expressions.

Logical operator. An operator that is used in a logical expression. The logical operators are: AND, OR, and NOT.

Loop. A sequence of instructions that is executed repeatedly until a terminating condition is reached. The FOR statement identifies the beginning of one type of

loop; the NEXT statement identifies its end. The DO statement identifies the beginning of another type; the LOOP statement identifies its end.

(B) Machine infinitesimal. The smallest positive and negative value which can be represented and manipulated by a BASIC implementation.

(B) Machine infinity. The positive and negative values of greatest magnitude which can be represented and manipulated by a BASIC implementation. It is not required that manipulations of machine infinity yield noninfinite results.

Main program. The first program unit to receive control when an IBM BASIC program is executed. The main program may invoke subprograms but cannot be invoked by them.

Mantissa. In floating-point notation (floating-point format), the number that precedes the E. The value represented is the product of the mantissa and that power of ten specified by the exponent.

*** Matrix (mathematical).** A rectangular array of elements, arranged in rows and columns, that may be manipulated according to the rules of matrix algebra.

Multiline function. A user-defined function that is defined with more than one statement.

Native format file. A file created by an IBM BASIC WRITE statement, containing user-defined data, organized sequentially or by record number or by key.

Nesting. (1) The occurrence of one or more loops within another loop. (2) The occurrence of a GOSUB statement when one or more GOSUB statements are already active. (3) The use of more than one set of parentheses to indicate the order of evaluation in a complex arithmetic expression.

Nonexecutable statement. A statement that is used to specify information to a compiler, but that does not explicitly result in executable code; for example, a declaration.

Null character string. An empty character string, usually specified by two adjacent single or double quotation marks.

Numeric character. Any of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Numeric constant. See Arithmetic constant.

Numeric expression. See Arithmetic expression.

Numeric operator. A symbol representing an operation to be performed upon arithmetic data. The numeric operators are:

+	addition and unary plus sign
-	subtraction and unary minus sign
*	multiplication
/	division
^ or ¬ or **	exponentiation or involution

Numeric variable. See Arithmetic variable.

Operand. A constant, a variable, an array member reference, a function reference, or a subexpression on which an operation is to be performed.

Operator. A symbol specifying an operation to be performed. See also Numeric operator, Binary operator, Concatenation operator, Logical operator, Relational operator, and Unary operator.

Output. The transfer of data from internal storage to an external medium.

Output list. A list of variables from which values are written to an output file; the list can be made up of scalar expressions and array references.

(B) Overflow. With respect to numeric operations, the term applied to the condition which exists when a prior operation has attempted to generate a result which exceeds machine infinity.

With respect to string operations, the term applied to the condition which exists when a prior operation has attempted to generate a result which has more characters than can be contained in a string of maximal length, as determined by the language processor.

With respect to string assignment, the term applied to the condition which exists when a prior operation has attempted to assign a value that is longer than the declared maximum of a string-variable or string-defined-function.

Padding. The addition of one or more blanks to the right or left of a character string to extend the string to a required length.

Parameter. A simple variable or empty array declarator enclosed in parentheses in a DEF statement or a SUB statement, and then used within that function or subprogram. The function or subprogram performs its calculations on the values substituted for the parameters when the function or subprogram is called.

Precision. The number of digits for which significance can be expressed, or the accuracy with which a number can be presented.

(B) Print zone. A contiguous set of character positions in a printed output line which may contain an evaluated print-statement element.

Priority. A rank assigned to a numeric operator; it is used when evaluating an arithmetic expression. The order of priorities, from high to low, is: exponentiation, unary operations, multiplication and division, addition and subtraction. Operations at the same priority level are evaluated as they are encountered (from left to right in the expression).

Program. A logically self-contained sequence of BASIC statements that can be executed by the computer to attain a specific result.

Programmer-defined function. See User-written function.

Program segmentation statement. Statements that provide a means to pass parameters between separately assembled programs or program segments. The statements SUB, SUBEXIT, ENDSUB, and CALL are program segmentation statements.

(B) Program Unit. A self-contained part of a BASIC program consisting of either the main program, which is a sequence of lines up to and including a line containing an END statement, or a subprogram, external function definition, picture definition, or parallel-section external to the main program.

Record. A collection of related data items treated as a unit.

Record-oriented file. A file in which items are stored in records.

Redimension specification. The assignment of a new dimension specification to an already existing array, via an array assignment statement, a READ statement, an INPUT statement, a GET statement, a READ File statement, or a REREAD File statement.

Redimensioning. The changing of the number of dimensions or the number of members in each dimension of a previously declared array.

Relational expression. An expression which uses relational operators to compare two numeric expressions or two character expressions.

Relational operator. An operator used in a logical subexpression. The relational operators are:

EQ or =	equal to
NE or <>	not equal to
GT or >	greater than
LT or <	less than
GE or >=	greater than or equal to
LE or <=	less than or equal to

Relative-record file. A file whose records are loaded into fixed-length, fixed-location slots.

Relative-record number. A number that identifies not only the slot in a relative-record data set but also the record occupying the slot.

Remark. See Comment.

Replication factor. The number of times a single data FORM specification is to be used.

Reserved word. A word that may not be used as a variable name, line label, function name, or subprogram name. (See Keyword.)

(B) Rounding. The process by which a representation of a value with lower precision is generated from a representation of higher precision taking into account the value of that portion of the original number which is to be omitted.

Scalar. A single data item (as opposed to an array of items).

Scalar expression. An arithmetic expression or a character expression representing a single value rather than an array of values.

Sequential access. The retrieval of data according to the order in which the data is stored in a file.

(B) Significant digits. The contiguous sequence of digits between the high-order nonzero digit and the low-order digit, without regard for the location of the radix point. Commonly, in a normalized floating-point internal representation, only the significant digits of a representation are maintained in the significand.

Simple variable. A scalar variable (but not an array member).

Single-line function. A user-defined function that is defined in one statement (that is, the DEF statement).

Slot. The space for a data record in a relative-record data set.

Source file. A sequentially accessible file containing EBCDIC coded records of BASIC language statements.

Special characters. Any characters allowed in IBM BASIC that are not alphabetic characters or digits.

Statement number. The number which prefaces an IBM BASIC statement. It can be up to seven digits in length (in the range 1 to 9999999).

Stream-oriented file. A file in which items are stored as a stream of data items and retrieved in sequential order.

Subprogram. A program unit beginning with a SUB statement and ending with an END SUB statement. Control is transferred to a subprogram by a CALL statement. Control is returned by the SUBEXIT statement.

Subroutine. A program segment (sequence of statements) branched to by a GOSUB statement. The last statement of a subroutine must be a RETURN statement which directs the computer to return and execute the statement following the GOSUB statement.

Subscript. Any valid arithmetic expression (whose rounded integer value is greater than or equal to zero) used to refer to a particular member of an array.

Substring. A part of a character string.

Terminal. A device, usually equipped with a keyboard and some kind of display, capable of sending and receiving information over a communication channel.

Text file. A file of internally coded data records suitable for input to a linkage editor or loader.

*** Truncation.** The deletion or omission of a leading or of a trailing portion of a string in accordance with specified criteria.

Unary. Having or consisting of a single component, element, or item.

*** Unary operator.** A numeric operator having only one term. The unary operators that can be used in absolute, relocatable, and arithmetic expressions are: (positive) + and (negative) -

(B) Underflow. With respect to numeric operations, the term applied to the condition which exists when a prior operation has attempted to generate a result, other than zero, which is less in magnitude than machine infinitesimal.

User. Anyone utilizing the services of a computing system.

User-written function. A function defined by the user in a single-line or multiline function definition.

Variable. A data item whose value may change during execution of a program.

Variable name. A name of a variable. The name consists of up to 40 alphabetic characters, optionally followed by a #, %, or $.

Vector. (1) * A quantity usually represented by an ordered set of numbers. (2) A collection of array data having a single dimension.

VS. Virtual Storage.

VSAM. Virtual Storage Access Method.

Workspace. The area which contains the program currently under development.

Zero suppression. The elimination of leading nonsignificant zeros in a number.

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W

Z

Reader's Comment Form

IBM BASIC
Application Programming:
Language Reference
GC26-4026-0

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL _____

Previous TNL _____

Previous TNL _____

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

GC26-4026-0

Reader's Comment Form

Fold and tape | **Please do not staple** | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

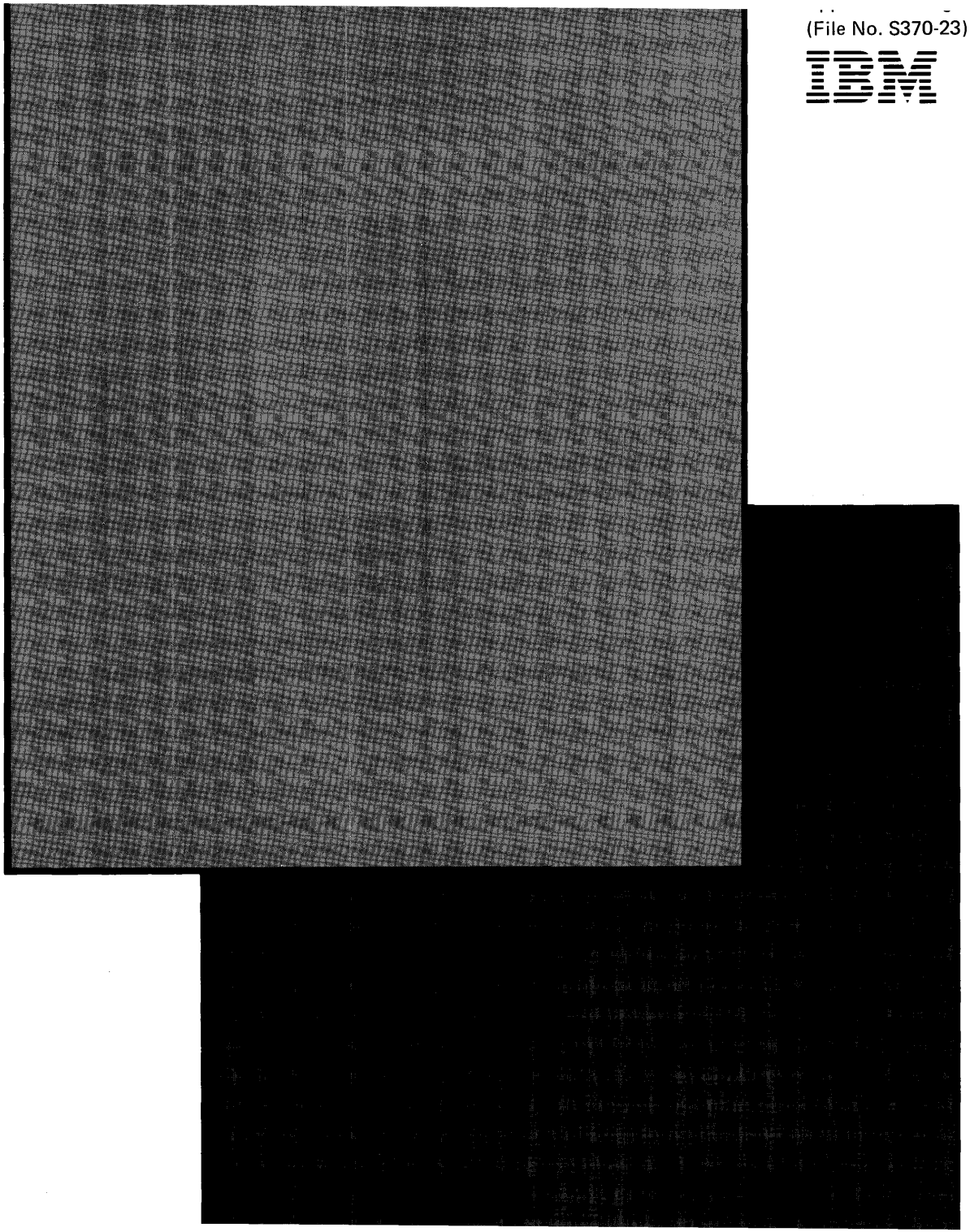
POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | **Please do not staple** | Fold and tape

IBM®

IBM BASIC Application Programming: Language Reference (File No. S370-23) Printed in U.S.A. GC26-4026-0



(File No. S370-23) Printed in U.S.A. GC26-4026-0

IBM