IBM BASIC Language Reference

Program Product

B B B B B

A A A A A

S S S S S

I I I I I

C C C C C

Order Number:
GC26-4026-2

Program Numbers:
5668-996 (VM)
5665-948 (MVS)

Release Number:
2

IBM

IBM BASIC Language Reference

Program Product

Order Number:
GC26-4026-2

Program Numbers:
5668-996 (VM)
5665-948 (MVS)

Release Number:
2

Third Edition (February 1986)

This is a major revision of, and makes obsolete, GC26-4026-1.

This edition applies to Release 2 of IBM BASIC/VM, Program Product 5668-996, Release 2 of IBM BASIC/MVS, Program Product 5665-948, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Amendments" following the section About This Manual. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, 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.

The IBM BASIC Processor and Library are available as two program products:

- IBM BASIC/VM, which runs under the Virtual Machine/System Product-Conversational Monitor System (VM/SP CMS), batch and interactive.
- IBM BASIC/MVS, which runs under the Multiple Virtual System/System Product batch, and with the Time Sharing Option (TSO) or the Time Sharing Option/Extended (TSO/E), batch and interactive.

IBM BASIC is used when referring to the common features of both products.

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 October 1984:

- 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

This manual should be used with the following IBM BASIC manuals:

IBM BASIC Programming Guide, SC26-4027, gives a step by step guide to practical programming.

IBM BASIC/VM System Services, SC26-4028, provides system-specific information for using IBM BASIC under VM/SP CMS.

IBM BASIC/MVS System Services, SC26-4106, provides system-specific information for using IBM BASIC under MVS and TSO.

IBM BASIC Language Reference Summary, SX26-3736, is a convenient, pocket-sized, quick reference card that summarizes IBM BASIC language elements.

To do VSAM file processing, refer to the following manuals:

For IBM BASIC/VM users:

VSE Virtual Storage Access Method: Programmer's Reference, SC24-5145

For IBM BASIC/MVS users:

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 manuals will be useful:

Graphical Data Display Manager Base: Programming Reference, SC33-0101

Graphical Data Display Manager Presentation Graphics Feature: Programming Reference, SC33-0102

Graphical Data Display Manager Presentation Graphics Feature: Interactive Chart Utility; User's Guide, SC33-0111

If BASIC programs are to communicate with IBM Database 2 (DB2) or Structured Query Language/Data Systems (SQL/DS), you will find the following manuals useful.

(Under MVS):

IBM Database 2 Introduction to SQL, GC26-4082

IBM Database 2 Application Programming Guide for TSO Users, SC26-4081

IBM Database 2 Messages and Codes, SC26-4113

(Under VM):

SQL/Data System Application Programming, SH24-5018

SQL/Data System Messages and Codes, SH24-5019

To use a 3800 type printer, refer to the following manual:

IBM 3800 Printing Subsystem Programmer's Guide, GC26-3846

If BASIC programs are to communicate with FORTRAN, COBOL, or PL/I, you will find the following manuals useful.

VS FORTRAN Language and Library Reference, SC26-4119

IBM VS COBOL for OS/VS, GC26-3857

OS and DOS PL/I Language Reference Manual, GC26-3977

Summary of Amendments

IBM BASIC/MVS and IBM BASIC/VM Release 2, February 1986

Enhancements

Release 2 includes many enhancements designed to make BASIC even easier to use than it was in Release 1. The principal enhancements are:

Full-Screen Editing: You can now make changes to any program line that appears on your screen.

Real Data: There are two additional numeric data types: real single and real double. Mathematical computations performed using these data types execute significantly faster than those using BASIC's decimal type. The use of real data also makes BASIC's interface to FORTRAN and PL/I much more useful.

MVS/XA Support: MVS/XA support makes very large programs and data arrays possible. BASIC can reside above the 16-megabyte line and use 31-bit addressing. BASIC can also be installed in the extended link pack area.

Relational Data: You can now access data managed by IBM's relational data base systems (DB2 and SQL/DS) with the CALL SQL interface. This brings the power and flexibility of a relational data base to the BASIC user.

GDDM Interface: You can now pass GDDM function names as well as RCP codes. This makes BASIC's GDDM calls easier to use. In addition, CALL GDDM has been enhanced to permit use of the Interactive Chart Utility (ICU), which is also supported both by name and with an RCP code.

FORTRAN, COBOL, and PL/I Interfaces: Numeric and character arrays can now be passed to subprograms written in these languages.

PF Keys in Command Mode: PF keys can be set to execute frequently used commands.

User Profile: You can specify commonly used function key settings and program options in a special profile data set. The profile is then processed automatically when you invoke BASIC.

Listing errors: It is now easier to validate programs, and detect and diagnose errors.

Highlighting the Prompt Line: This increases emphasis on the user input area.

Grouping help panels for printing: Allows users to print subsets of the many available help panels.

Tape Support: IBM BASIC/VM now supports multi-file tapes.

Changes to this Manual

To make this manual easier and more useful, the following changes have been made:

- Many new examples have been added.
- The glossary and index have been updated.
- The manual has been totally reorganized.

Migration from Release 1

IBM BASIC Release 2 is a functional extension of Release 1. With only minor exceptions, all programs which run correctly on Release 1 will also run correctly on Release 2.

The minor exceptions which need to be considered in migrating from Release 1 to Release 2 are the following:

- Keywords: There are nine new reserved keywords (ARITHMETIC, BASIC, DBL, DOUBLE, PARM$, REAL, SINGLE, SNG, and SQL).
- Exception Codes: There are several new and modified exception messages and codes, but they should have little or no impact on Release 1 programs.
- OPTION ARITHMETIC: The new REAL data type should not impact operation of programs created in Release 1 if the IBM-distributed default (OPTION ARITHMETIC DECIMAL) is used. If OPTION ARITHMETIC NATIVE is specified, default typing will be REAL, program identifiers ending with # will be invalid in a DECIMAL statement, and the program may produce different results.
- **Using programs created with Release 1:**

  Release 1 programs may have been COMPILEd, SAVEd, or STOREd. Any and all of these programs may be easily and successfully used in the Release 2 environment:

  - Programs COMPILEd with Release 1 will run without change in Release 2.
  - Programs SAVEd in Release 1 may be LOADed and used without change in Release 2 (except for reserved keywords noted above).

- Programs STOREd in Release 1 may be FETCHed and used in Release 2. FETCH in Release 2 is modified to be able to read files which were STOREd under Release 1. This is important, because Release 2 internal text files have a different format than Release 1 files. When the Release 2 FETCH encounters a back level (Release 1) STOREd file it automatically converts the data to the new Release 2 internal format. A subsequent STORE will save the new internal text format.

- IBM BASIC Library modules for Release 1 and Release 2 are different. The Release 1 and Release 2 Library modules may not be mixed; they must be either all Release 1 or all Release 2.

- Release 2 object (compiled) code will not necessarily work correctly with Release 1 Libraries.

- **FONT:** In Release 1, when a line is printed, the last FONT is used even if the data was formatted in a prior PRINT statement with a different font. In Release 2, the font specified for each PRINT statement in which the data was formatted is used.

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. IBM BASIC also provides versatility of use; the language can be used in two different ways:

- INSIDE THE BASIC ENVIRONMENT, you can create and execute programs using the full range of IBM BASIC's editing capabilities. You can also execute programs that have already been compiled.

- OUTSIDE THE BASIC ENVIRONMENT, you can separately compile IBM BASIC programs and then execute the compiled programs under direct operating system control. This is similar to other language processors such as FORTRAN or COBOL.

The IBM BASIC program product consists of a Processor and a Library. If you are inside the BASIC environment, you need both the Processor and the Library. Outside the BASIC environment, if you are compiling, you also need both the Processor and the Library. If, however, you are only running programs outside the BASIC environment, you need only the Library.

IBM BASIC/MVS is both source and object code compatible with IBM BASIC/VM. Valid source programs and compiled object code that run on either MVS/SP or VM/SP CMS will run on the other. Changes are necessary only for any system-dependent functions invoked in the CALL SYSTEM statement, for any system-dependent file names in the OPEN and CHAIN statements, and for system-dependent exception codes.

How to Use this Book

The *IBM BASIC Language Reference* is divided into two parts. The first part contains information about the BASIC language and BASIC statements. It includes discussions of data types and variables, numeric and character expressions, functions, file capabilities, and general information on BASIC statements, immediate statements, and commands.

Part Two: Statements, Functions, and Commands, beginning on page 143, contains detailed descriptions of the IBM BASIC statements, intrinsic functions, and commands. In addition to an explanation of the use of the statement, intrinsic

function, or command, each description contains the syntax, and an example when it is appropriate.

This book is intended to be used for reference. If you want more information on BASIC programming, see the *IBM BASIC Programming Guide*.

The IBM 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

Inside the BASIC Environment

Inside the BASIC environment, you can create, edit, debug, and run programs using an interactive terminal. The programs themselves can also interact directly with your terminal.

The BASIC environment provides editing facilities and a set of *commands*. These commands instruct BASIC to perform the following functions:

- Create and edit program lines
- Load, merge, or save programs
- Start and control execution of programs
- Display information about the BASIC language and BASIC files
- Delete files
- Compile programs

For more information on commands and editing, see "BASIC Commands and Editing Facilities" on page 137.

Inside the BASIC environment, several statements can be used in either of two ways. They can be used within programs, as usual, or can be executed immediately to act like commands. With such *immediate statements*, you can perform the following operations:

- Print variables, arrays, or expressions
- Assign values to variables or arrays
- Calculate using intrinsic functions and numeric, character, and array operations
- Specify immediate options
- Declare immediate variables

Immediate statements are particularly useful in debugging. They allow you to inspect and change variables within a program while execution of that program has been suspended. For more information on immediate statements, see "Immediate Statements" on page 133.

Outside the BASIC Environment

Outside the BASIC environment, you can both compile programs (with the BASIC command) and execute precompiled programs (with the BASICRUN command).

You invoke BASIC as a compiler directly from the host system for each source program file to be compiled. You can elect to produce an object program file and a listing file.

You can also compile and execute BASIC programs using your host operating system's batch capabilities.

Compiling and executing outside the BASIC environment are system dependent. For details, see the IBM BASIC System Services manual for your system (VM or MVS).

Part One: The Elements of BASIC

Structure of a Basic Program

Character Set

The character set is:

Character	Meaning
A - Z	Uppercase letters
a - z	Lowercase letters
0 - 9	Digits
	Space (blank)
&	Ampersand
'	Apostrophe
*	Asterisk
:	Colon
,	Comma
$	Dollar sign
=	Equal sign
!	Exclamation mark
¬ or ^	Exponentiation symbol (NOT sign or circumflex)
>	Greater than sign
(	Left parenthesis
<	Less than sign
-	Minus sign
#	Number sign
%	Percent sign
.	Period
+	Plus sign
"	Quotation mark
)	Right parenthesis
;	Semicolon
/	Slash
_	Underline

Note: Except within character strings, uppercase and lowercase letters are equivalent in a BASIC program.

Additional characters are allowed in character strings, comments, and external files.

Identifiers

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

Identifiers used to name variables, arrays, and user-defined functions are local to the program unit in which they occur; they may be used to name different objects in different program units. Identifiers used to name subprograms are global to the entire program; they name the same subprogram wherever they occur.

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

The first character of an identifier must be a letter, which may be followed by any combination 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 for a variable, array, or function 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 31.

An identifier for a subprogram or a line label may *not* end with a number sign, percent sign, or dollar sign.

Examples (correct identifiers)

```
ALPHA$
Alpha$
alpha_betics$
DEC_NUM
INT_num%
decNum#
```

Examples (incorrect identifiers)

```
THIS_IS_A_SAMPLE_IDENTIFIER_WHICH_IS_TOO_LONG
     (more than 40 characters)

SUB SORT_THE_LIST
     (subprogram name has more than 7 characters)

SUB DOIT%
     (subprogram name may not end with a %)

!IDNAME
     (does not start with a letter)

VARIABLE NAME
     (spaces not allowed)
```

A given identifier may name a variable, an array, ***or*** a function, but once that identifier is used to name one of those three, it cannot be used again as a variable, array, or function name within that program unit.

It ***can*** also be used, however, as either a subprogram name or a line label within that same program unit, provided that it meets the requirements for such identifiers as listed above. Context always determines how it is interpreted. You may, however, find that using the same identifier in more than one way is confusing.

Language Statements

A BASIC source language line contains a line number, an optional line label, and one or more statements.

Lines and Line Numbers

A BASIC program consists of a series of lines. In general, lines cannot contain more than 156 characters. (For the exception, see "Continuation of Statements" on page 16.)

Each line starts with a unique number. The smallest line number allowed is 1 and the largest is 9999999.

The line numbers will always determine the sequence of lines in the program. Regardless of the order in which lines are entered, they will be sorted in ascending line number order. For this reason, duplicate line numbers would cause an ambiguity and are not allowed. Therefore, if you enter a line that has the same line number as an existing line, the line you just entered will replace the existing line.

Line numbers also provide labels for lines.

Example

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

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

Line Labels

Line labels can be used in lieu of line numbers as branch destinations. A line label is declared by its appearance after a line number; the line label must be followed by a colon (:).

Only one line label may be declared for any one line number. If a line label is declared for a line number, the same line label must not be declared for another line in the same program unit.

Line labels may be up to 40 characters long, the first of which must be alphabetic (A through Z, upper or lowercase). The remaining characters may be either alphabetic, numeric, or the underline character. Certain identifiers may not be used as line labels (see "Reserved Words" on page 12 and "Keywords" on page 10 for restrictions).

Example

```
100 GOTO FIRST_CHOICE
.
.
.
700 FIRST_CHOICE: LET A=B
```

In this example, the GOTO statement at line 100 directs processing to the line whose label is FIRST__CHOICE.

Keywords

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

The first keyword of a statement indicates the action to be performed by the statement (READ, WRITE, and so forth).

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

Example

```
LET
Let
let
```

are equivalent.

Except for the following, keywords may not be abbreviated.

- COM may be used instead of COMMON
- REC may be used instead of RECORD
- PAGE may be used instead of NEWPAGE

Keyword List

The IBM-distributed reserved words are identical to the keywords. Check with your system administrator to find out whether you installation has a different reserved word list.

The following is a list of BASIC keywords in alphabetical order.

ABS	ASORT	CALL	CNT
ACCESS	ATN	CASE	COBOL
ACOS	ATTN	CAUSE	CODE
AIDX		CEIL	COLLATE
AND	BASE	CEN	COM
ANGLE	BASIC	CHAIN	COMMON
APPEND	BEGIN	CHR$	CON
ARITHMETIC	BOTTOM	CLINK	CONTINUE
ASIN	BREAK	CLOSE	CONV

COS
COSH
COT
CSC

DATA
DATE
DATE$
DAT$
| DBL
DEBUG
DEC
DECIMAL
DEF
DEFDBL
DEFINT
DEFSNG
DEG
DELETE
DET
DIDX
DIM
DISPLAY
DO
DOT
| DOUBLE
DSORT
DUPKEY
DUPREC

ELSE
END
ENDPAGE
EOF
EPS
EQ
ERR
ERROR
EXIT
EXP

FAH
FIELDS
FILE
FILENUM
FILES
FILE$
FIPS
FIXED
FLAG
FLINK
FNEND
FONT
FOR
FORM
FORTRAN
FP

GDDM
GE
GET
GO
GOSUB
GOTO
GT

IDN
IF
IFIX
IGNORE
IMAGE
INF
INPUT
INT
INTEGER
INTERNAL
INV
INVP
IOERR
IP

JDY

KEY
KEYED
KEYNUM
KLN
KPS

LE
LEFT
LEN
LET
LINE
LINPUT
LOG
LOG2
LOG10
LOOP
LPAD$
LPREC
LT
LTRM$
LWRC$

MARGIN
MAT
MAX
MIN
MOD

NATIVE
NE
NEWPAGE
NEXT
NOFIPS
NOKEY
NONE
NOREC
NOT
NUL$

OFF
OFLOW
ON
OPEN
OPTION
OR
ORD
ORGANIZATION
OUTIN
OUTPUT

PAGE
PAGEOFLOW
| PARM$
PAUSE
PI
PLI
PLINK
POINTER
POS
PRD
PRINT
PROMPT
PRTZO
PUT

RAD
RANDOMIZE
RD
READ
| REAL
REC
RECORD
RECORDS
RELATIVE
REM
REREAD
RESET
REST
RESTORE
RETRY
RETURN
REWRITE
RIGHT
RLN
RND
ROUND
RPAD$
RPT$
RTRM$

SCRATCH
SEARCH
SEC
SELECT
SEQUENTIAL
SGN
SIN
| SINGLE
SINH
SIZE
SKEY
SKIP
| SNG
SOFLOW
SPREC
| SQL
SQR
SRCH
SREP$
STANDARD
STEP
STOP
STREAM
STR$
SUB
SUBEXIT
SUM
SYSTEM

TAB
TAN
TANH
THEN
TIME
TIME$
TO
TOP
TRACE
TRN

TRUNCATE	UFLOW	USING	WHILE
TYPE	UNTIL		WRITE
	UPRC$	VAL	
UDIM	USE	VARIABLE	ZDIV
			ZER

Reserved Words

Reserved words are words you cannot use as identifiers.

The IBM-distributed reserved word list is the same as the keyword list (see "Keyword List" on page 10), but your organization may have customized your reserved word list by adding or removing reserved words. Check with your system administrator to see whether your list has been changed.

Example

```
10 LET LET = 2
20 LET = 2
```

Neither statement is valid, because LET is a reserved word.

Rules for Keywords Removed from the Reserved Word List

If your organization has removed keywords from the BASIC reserved word list, the rules regarding treatment of keywords are different from those given in "Reserved Words" and "Keywords" on page 10. The rules for keywords which have been removed from the reserved word list are as follows:

1. If the keyword is listed in the "Keyword List" on page 10, but is neither an intrinsic function name nor an array function name listed in "List of Intrinsic Functions and Array Functions" on page 380, then:

   - Where the keyword can legitimately appear as a keyword, it is treated as the keyword. Otherwise, it is treated as an identifier. The characters preceding the keyword and the next nonblank character following the keyword are used to determine if the keyword may appear. If an ambiguity is detected, the keyword interpretation is used.

2. If the keyword is an intrinsic function name or an array function name listed in "List of Intrinsic Functions and Array Functions" on page 380, the same rules apply except that:

   - If the keyword is declared in a COMMON, DECIMAL, DEF, DEFDBL, DEFINT, DEFSNG, DIM, INTEGER, REAL, or SUB (as a parameter) statement, the keyword will be treated as an identifier throughout the program unit, and not as a function.

   - If the keyword is not declared explicitly as described above, and the first occurrence causes the keyword to be treated as an identifier, then the keyword will be treated as an identifier throughout the program unit. Otherwise, the keyword is treated as the intrinsic or array function throughout the program unit.

Example

```
10 LET LET = 2
10 LET = 2
```

Both statements above are valid if LET is not included in the reserved word list, although it is still an IBM BASIC keyword. In this example, both of the LET statements 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.

In some situations, keywords may be ambiguous if they are not reserved; therefore, whenever an ambiguity is detected, the word is interpreted as a keyword rather than an identifier.

Example

```
100 REM = 3
200 IMAGE: X=Y
300 X = SIN(Y)
400 Y = COS
```

In the above example:

- Statement 100 is interpreted as a REM statement, not an assignment statement.
- Statement 200 is interpreted as an IMAGE statement, not an assignment statement with a line label named IMAGE.
- Statement 300 is a reference to the intrinsic function SIN, not an implicit declaration of a numeric array SIN, unless SIN has been explicitly declared to be an array in the program unit, in which case it is a reference to a numeric array SIN. An error will occur if an explicit or implicit declaration on a previous line has declared that SIN is a variable.
- Statement 400 is a reference to the variable COS, not a reference to the function COS. An error will occur if an explicit or implicit declaration on a previous line has declared that COS is not a variable.

Meaningful Spaces

Spaces (blanks) cannot appear within:

- Line numbers
- Keywords
- Identifiers
- Numeric constants

Example

```
BOT TOM
```

is not acceptable.

The only keywords which may contain spaces are GOTO and GOSUB. GOTO and GO TO are equivalent; GOSUB and GO SUB are equivalent.

When the presence of separating characters delimits keywords or identifiers, the keywords or identifiers can appear with or without separating spaces. In addition to the space, separating characters are:

```
+ - * / = < > " ' ¬ , ; : & ! ( )
```

Examples

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

```
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 separating character. If it is not at the end of a line, it must be followed by a space or other separating 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 (=).

Examples

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

are all equivalent.

Spaces appearing in quoted character constants (character constants enclosed in quotation marks or apostrophes) are treated as part of the constant; spaces appearing outside the quotation are not considered part of the constant.

Example

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

The space between T and Y is part of the string.

Spaces either preceding or following an unquoted character constant (only allowed in DATA statements and in responses to INPUT statements) are not considered part of the character string.

Example

```
400 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.

Comments

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

Note: You should not use any character in comments whose hexadecimal value is in the range 00 through 05. BASIC uses these values as internal flags; they will be changed to spaces.

The contents of your BASIC source program may be processed as data by software and hardware other than IBM BASIC (listing to a printer, for instance). Therefore, you should only use control characters (hexadecimal 06 through 3F, and FF) for which you know the effects in your software and hardware environment.

Exclamation Mark Comments

An exclamation mark (!) specifies that the remaining characters on the current line are a comment and are not to be interpreted; the characters are displayed in the program listing and no other action taken.

Examples

```
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
```

The comments document the purpose of these three statements.

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

Exclamation mark comments are sometimes referred to as trailing comments.

If the comment follows a statement to be continued, the line should end with an ampersand and the statement should be continued on the next line. The comment itself may not be continued.

REM Statement Comments

The keyword REM (for remarks) is used for comment statements. REM specifies that characters of the entire statement (to the end of the line) are a comment and not to be interpreted; characters are to be displayed in the program listing and no other action taken. An exclamation mark (!) may be used in place of the REM keyword.

Examples

```
100 REM BEGIN PROCESSING : PART 1
110 LET ABC = PARTA + PARTB : REM TOTAL THE PARTS
120 ! BEGIN PROCESSING : PART 2
```

Note that the colons (:) in lines 100 and 120 are part of the comment and do *not* indicate the end of the REM statement.

A REM statement (specified with either the keyword REM or with an exclamation mark) cannot be continued. If the comment ends with an ampersand, the ampersand is considered part of the comment, not an indication that the REM statement is to be continued.

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 after the leading line number and its following space where a space might normally appear, except within a character constant.

Line numbers are not allowed at the beginning of a physical line that is the continuation of a previous line.

```
100 A = B  &
200 & C = D
```

This violates IBM BASIC syntax.

Continuations are not allowed for REM (remarks) statements. Lines containing exclamation mark comments (!) may be continued, but the comment itself 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, but each continuation must be 156 characters or fewer.

Note that a statement that begins with an exclamation mark is considered to be equivalent to a REM statement and cannot, therefore, be continued.

Multiple Statements Per Line

Except as noted, the colon (:) indicates that another statement follows on the same line. The exceptions are when the colon is:

- Within a quoted character string
- Used in DATA statements
- Used in a substring qualifier
- Used to signify a line label
- Used in a file I/O or IMAGE statement
- Used in comments

A colon which follows a statement must be followed by another statement on the 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 could end with an optional colon and another statement is to follow on the same line, the optional colon must be included and the presence of a second colon is recognized as the indication that another statement is present on the same line. Statements with optional ending colons are CLOSE, DELETE, PRINT, PRINT FIELDS, PRINT File, RESET, RESTORE, and SCRATCH.

Example

```
100 PRINT #2:: PRINT #3: "THE ANSWER IS", A
```

is functionally equivalent to two separate PRINT statements:

```
100 PRINT #2:
110 PRINT #3: "THE ANSWER IS", A
```

where the first colon is part of the syntax of the PRINT #2: statement, and the second colon indicates that another statement begins on that line.

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

An END or END SUB statement may only be followed by a REM statement on the same line.

The DATA, EXIT, FORM, IF, IMAGE, and REM statements must be last on a line. In addition, a DATA statement must be the only statement on a line if it is referenced by a RESTORE statement with a line number or label.

In a REM statement, or in any statement containing an exclamation mark comment (!), no other statement can follow on the same line, because all characters following the keyword REM or the exclamation mark on the same line are taken to be a comment.

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 multi-statement 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 in the list of imperative statements following the THEN and ELSE of an IF statement. When an IF statement ends with a list of imperative statements (as opposed to a line number reference or a line label reference), all the remaining statements on the line are considered part of the IF statement. When an IF statement ends with *THEN line-ref* or *ELSE line-ref*, no other statements may follow on the line.

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 90
- Loop blocks—described in "Loop Control Statements" on page 74
- IF blocks—described in "IF Blocks" on page 77
- SELECT/CASE blocks—described in "SELECT Blocks" on page 79

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. Therefore, the same identifier may be used in different program units to name different items.

Statements within a program unit may not normally refer to any variable, array, line label, line number, or function (other than intrinsic functions) defined externally to that program unit. This local nature of variable identifiers can be changed by making specific variables *global* (known to other program units). (See "COMMON Statement" on page 170, "CHAIN Statement" on page 166 and "USE Statement" on page 373.)

Program units are described in "Subprogram Statements" on page 93.

Data: Constants, Variables, and Arrays

Data can be constants, variables, or arrays. Therefore, a constant, a variable name, or an array name may be specified to refer to data.

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

Constants

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

Numeric Constants

Numeric (also called arithmetic) constants can be represented in three forms: integer notation, fixed-point notation, and floating-point notation. The actual type of the constant is determined from its appearance, value, context, and from the options in effect, as described in "Determining the Type of Numeric Constants" on page 23.

A sign may optionally precede the constant and is considered to be part of the constant. If the constant is unsigned, it is assumed to be positive. (A sign preceding a zero constant has no effect.)

A numeric constant may *not* contain spaces.

Integer Notation

Integer notation represents constants which are whole numbers. They can be positive, zero, or negative.

Integer notation consists of one or more digits with no decimal point and no exponent. Periods, commas, and spaces must not appear in integer notation.

Examples (valid integer notation)

```
15
+365
-123
2147483647
```

Examples (invalid integer notation)

`1,234,567,890`	Commas not allowed
`$493`	Dollar signs not allowed
`394.31`	Periods not allowed
`400 320`	Spaces not allowed

A constant represented in integer notation will have integer type only if it is in the range :

```
-2147483648 to +2147483647
```

Otherwise, a constant represented in integer notation will have decimal or real type. The type is determined from the context and the options as described in "Determining the Type of Numeric Constants" on page 23.

Fixed-Point Notation

Fixed-point notation is used to represent constants with an explicit decimal point.

Fixed-point notation consists of one or more digits with a decimal point but no exponent.

Examples (valid fixed-point notation)

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

Examples (invalid fixed-point notation)

`1,234`	Commas not allowed
`$493.95`	Dollar signs not allowed
`1.23E4`	The letter E is not allowed

A constant represented in fixed-point notation will have decimal or real type. The type is determined from the context and from the options as described in "Determining the Type of Numeric Constants" on page 23.

Floating-Point Notation

Floating-point notation (also called scientific notation) allows the representation of very large or very small numeric values.

A constant can be written in floating-point notation as either an integer notation constant or a fixed-point notation constant, followed by the letter E, followed by an integer constant. The E stands for "times ten to the power" and the integer after the E denotes the power of 10 by which the leading numeric constant is to be multiplied. (The letter E may be either uppercase or lowercase.)

Examples (valid floating-point notation)

```
5.0E+6
5E6
+5e06
.05E8
```

Each of these examples represents the number 5 million, or 5 times 10 to the power 6.

Each of these examples specifies that the decimal point is to be moved right the number of places indicated by the number after the letter E (that is, 5 is to be multiplied by the 6th power of 10). The number before the E is called the mantissa. The number after the E is called the exponent.

Likewise, very small number values can be represented with a negative exponent (meaning move the decimal left) as follows:

Examples (valid floating-point notation)

```
5.0E-6
5E-06
+5E-6
.05E-4
```

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

Examples (invalid floating-point notation)

`1.5D6`	The letter D is not allowed
`2.E`	Requires an integer after the E
`-1,234E5`	Commas not allowed
`$49395e-2`	Dollar signs not allowed
`7 E10`	Spaces not allowed

A constant represented in floating-point notation will have decimal or real type. The type is determined from the context and from the options as described in "Determining the Type of Numeric Constants."

The absolute value of a constant must be in the range:

```
.1E-999 to .99999999999999999E+999
```

If the constant is too large or too small, a syntax warning message will be produced.

Determining the Type of Numeric Constants

Normally, numeric constants should be specified in the notation most natural to the user, without consideration for internal representation or data type. However, when numeric constants are used as arguments to subprograms, or in certain situations involving precision, range, expression evaluation, or where performance is critical, the data type associated with a constant can be significant.

The following rules, in order of precedence, are used to determine the data type of a numeric constant:

1. If the constant is represented in integer notation in the range:

   ```
   -2147483648 to +2147483647
   ```

   it will have integer type.

2. If the constant appears as the argument of one of the following typing functions, the resulting constant will have the type specified by the typing function as shown:

Function	*Type*
DEC	Decimal
SNG	Real Single
DBL	Real Double
REAL	Real Single if SPREC option is in effect; Real Double if LPREC option is in effect. See "OPTION Statement" on page 312 for information on options.

3. If the constant is used in an expression, the resulting constant will have the same type as the operand it is to be combined with, except:

   - when that operand has integer type, or
   - when the type of that operand has not been determined (for example, when that operand is also a constant), or
   - when there is no other operand.

4. If the constant is assigned in a simple LET statement which has only one variable or array element on the left side of the equal sign, the constant will have the same type as the variable or array element unless that type is integer.

5. If the constant is assigned in a LET statement which has more than one variable or array element on the left side of the equal sign, the constant will have the same type as the variable or array element on the left side with the highest type (unless integer). The priority of numeric types, from highest to lowest, is decimal, real double, real single, and integer.

6. If the optionally signed constant is the argument of a user-defined function and the corresponding parameter has integer type, the type of the constant is determined by the options in effect as defined above for a LET statement. If the corresponding parameter does not have integer type, the constant has the same type as the parameter.

7. If the above rules have not determined the type, then the type of the constant is determined by the options in effect:

   - ARITHMETIC DECIMAL makes the constant decimal
   - ARITHMETIC NATIVE and SPREC makes the constant real single
   - ARITHMETIC NATIVE and LPREC makes the constant real double

   See "OPTION Statement" on page 312 for more information on options.

If after determining the type of the constant, the value is too large or too small for that type, an exception (constant overflow or underflow) will occur when that constant is referenced during execution. See "Numeric Data Types" on page 26 for the range of the different types, the use of numeric types, and their 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 quotation marks or apostrophes (called delimiters). These delimiters may be omitted in DATA statements and in the responses to INPUT or LINE INPUT statements under specific rules which are discussed in the sections on the DATA and INPUT statements.

Character constants may be specified using either apostrophes or quotation marks as a delimiter for the constant. The delimiter which ends a character constant must be the same as the one that started it.

Examples

If this constant is specified:

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

the value is:

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

with quotation marks around the word Halt. If this constant is specified:

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

the value is:

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

with apostrophes around the word Halt.

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

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

If an apostrophe (the same character as a single quotation mark) appears in a character constant, it can be specified using either double quotation marks as the delimiter or a pair of single quotation marks with no intervening spaces:

The value:

```
The sentry's station
```

may be specified as either:

```
"The sentry's station"
```

or:

```
'The sentry''s station'
```

Similarly, if a quotation mark appears in a character constant, the constant may be specified using either apostrophes as the delimiter, or as a pair of quotation marks with no intervening spaces.

A character constant may be null (contain no characters). The null string (a string of length zero) may be specified with either contiguous apostrophes ('') or contiguous quotation marks ("").

See also "Character Data Type" on page 31 for the internal representation of character constants. For a character constant, the maximum length and the current length are equal to the number of characters in the constant (delimiter pairs inside the constant count as 1 character).

Data Types

The type of a constant, variable, or an array corresponds to the type of data the constant, variable, or array represents.

Data is divided into two classes: numeric and character. There are four types of numeric data: integer, decimal, real single, and real double. Character data is of the character type.

Numeric Data Types

Use of Numeric Data Types

It is usually not necessary to choose a specific numeric data type but there are cases where considerations of precision, range, or performance can make it desirable to specify a type.

Decimal data allows the greatest range of any BASIC numeric data type and is more precise than real data. Decimal data is normally the default data type in BASIC and can be used in most cases. However, integer and real data provide advantages for special situations.

Integer data provides the smallest range of any numeric data type. However, integer data are more efficient for use as index values (for arrays), when the range of integers is sufficient, and may be used in loops when speed is important.

Computation using real data is more efficient than with decimal data because it uses the computer's floating-point format. If your program does lots of computation, use real data (or integer where appropriate) to allow your program to run faster.

When your BASIC program passes data to other language subprograms, or to GDDM, you may also save time by using real or integer data instead of decimal. Since decimal data is stored in a format that is internal to BASIC, a decimal data item is converted to real when it is passed to another language subprogram. The conversion step can be avoided if you define the data to be real in your BASIC program.

Real single data should be used in preference to real double when storage space is a consideration. Real double data should be used in preference to real single data when accuracy is a consideration.

Real data is not as accurate as decimal data and rounding errors may occur. For example, the value 0.1 cannot be exactly represented using real data. It may be displayed by a PRINT USING statement as:

```
0.100000023841857910    if type is real single
0.099999999999999992    if type is real double
0.100000000000000000    if type is decimal
```

Integer Data

The integer data type is useful for index values and other cases where whole numbers are normally used.

The range of integer data is:

```
-2147483648 to +2147483647
```

Internal Representation of Integer Data: Integer data are stored on a fullword boundary as fullword (32-bit) twos-complement values as shown in Figure 1.

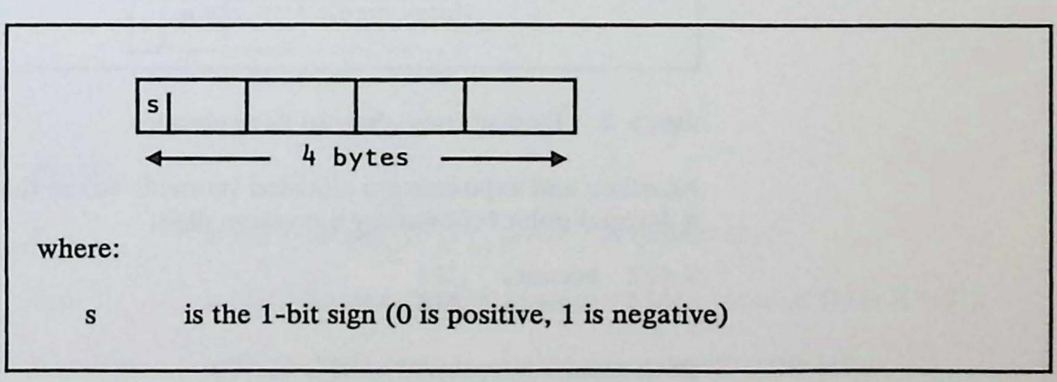
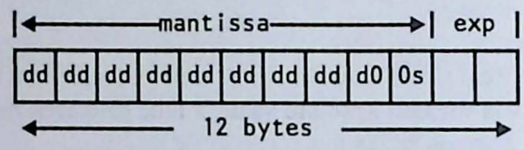


Figure 1. Integer Data—Internal Representation

The decimal data type can be used for all numeric data in BASIC. It is normally the default data type and the easiest to use.

Internal Representation of Decimal Data: Decimal data are stored internally on a fullword boundary as three words (12 bytes) as shown in Figure 2. A zero value is represented by 3 words of zeros.



where:

mantissa — is a 10-byte packed decimal number normalized to the left with the last two digits zero.

d — represents a half byte containing the value of a decimal digit (0 through 9).

s — represents the half byte for the sign:

B'1010' positive
B'1011' negative
B'1100' positive (preferred)
B'1101' negative (preferred)
B'1110' positive
B'1111' negative

exp — represents the half word (power of 10) exponent as a twos-complement value.

Figure 2. Decimal Data—Internal Representation

Mantissa and exponent are adjusted (normalized) so that the mantissa begins with a decimal point followed by a nonzero digit:

```
5.0E2  becomes  .5E3
.05E4  becomes  .5E3
```

Both numbers represent a value of 500.

The mantissa may contain up to 17 digits. Note that the internal format would allow up to 19 digits but only 17 digits (after rounding) are stored. The other two digits are used to maintain accuracy in intermediate results of computation of an expression.

The exponent can be in the range -999 to +999. Therefore, the largest absolute value is:

```
.99999999999999999E+999
```

and the smallest absolute value is:

```
.1E-999
```

Note: The largest absolute value will be rounded and displayed by the PRINT statement as:

```
1.E+999
```

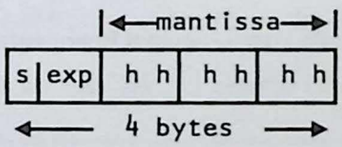
The smallest absolute value will be displayed by the PRINT statement as:

```
1.E-1000
```

Real Single Data

The real single data type is the System/370 single precision floating-point format, in hexadecimal. It approximates the value in 4 bytes of storage. It is useful when speed is important and the range and precision of decimal are not required. It is not as precise as real double data, but requires half the space.

Internal Representation of Real Single Data: The real single data type is floating-point binary data with a precision of (approximately) 6 decimal digits and an exponent in the range of (approximately) -78 to +75. It is stored internally in S/370 floating-point format as a 1-bit sign, a 7-bit biased hexadecimal characteristic (exponent), and a 6-digit hexadecimal fraction (mantissa). Hence, each real single datum occupies a full word (32 bits) of storage on a fullword boundary as shown in Figure 3.



where:

s	is the 1-bit sign (0 is positive, 1 is negative).
exp	is the 7-bit hexadecimal (power of 16) exponent (bias X'40').
mantissa	is 3-byte hexadecimal number normalized to the left.
h	is hexadecimal digit (0-F).

Figure 3. Real Single Data—Internal Representation

The real double data type is the System 370 double precision floating-point format, in hexadecimal. It approximates the value in 8 bytes of storage It is useful when speed is important and real single does not satisfy the need for accuracy, but the range and precision of decimal are not required. It is more precise than real single data, but requires twice the space.

Internal Representation of Real Double Data: The real double data type is floating-point binary data with a precision of (approximately) 15 decimal digits and a scale of (approximately) -78 to +75. It is stored internally in S/370 floating-point format as a 1-bit sign, a 7-bit biased hexadecimal characteristic, and a 14-digit hexadecimal fraction. Hence, each real double datum occupies a double word (64 bits) of storage on a fullword boundary as shown in Figure 4.

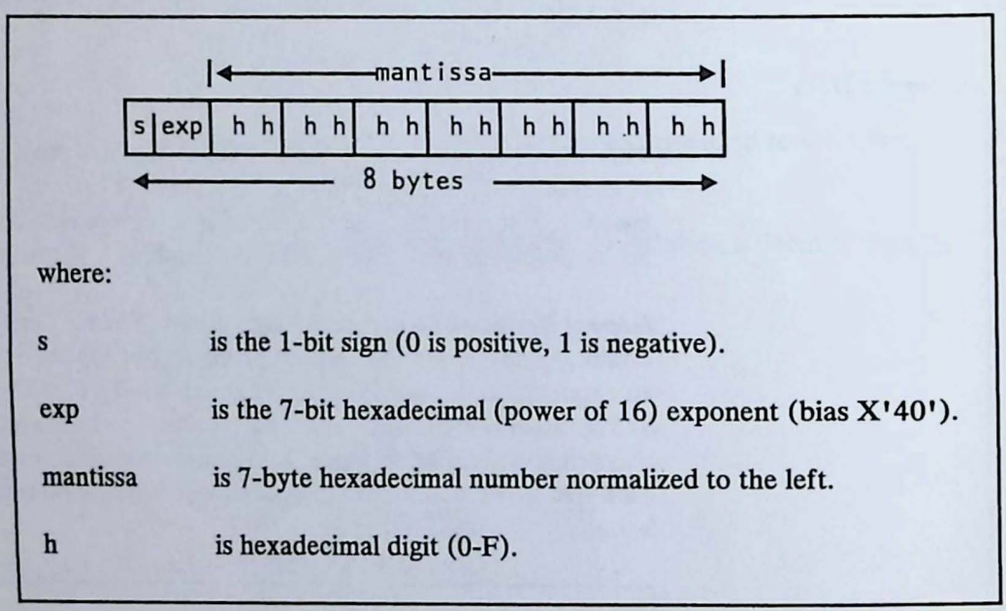


Figure 4. Real Double Data—Internal Representation

Character Data Type

Character data are variable-length strings of characters.

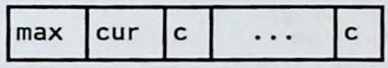
The maximum length of a character data item is 32767.

Note: Character data may be processed by software and hardware other than IBM BASIC, for example, when sending a file to the printer. Therefore, you should avoid using strings that contain control characters (hexadecimal 00 through 3F, and FF), or use only control characters for which you know the effects in your software and hardware environment.

In character constants of your BASIC source program, you should not use any characters whose hexadecimal value is in the range 00 through 05. BASIC uses these values as internal flags. (If you do, they will be changed to spaces.)

On output to 327x type terminals in scroll mode, control characters (01 through 3F) are converted to spaces.

Internal Representation of Character Data: Character data are stored on a fullword boundary as shown in Figure 5.



where:

max is the maximum length of the data item (2 bytes).

cur is the current length of the data item (2 bytes).

c is an EBCDIC character (byte).

Figure 5. Character Data—Internal Representation

Variables

Numeric data may be retained and referenced by a numeric variable name. 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 element) of an array.

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. The characters %, # and $, which denote the data type of the variable, are considered as part of the 40 characters when determining variable name size limits.

Certain identifiers may not be used as variable names (see "Reserved Words" on page 12).

Numeric Variables

Numeric variables:

- Are decimal, integer, real single, or real double type .
- Are specified explicitly by type with the DECIMAL, INTEGER, REAL, DEFINT, DEFSNG, and DEFDBL statements.
- Are specified implicitly by type with the suffix % or # on the variable name.
- Are assigned a type by default in the absence of explicit or implicit typing.
- Are in error if the explicit and implicit specifications conflict.
- May not have names ending with $.
- Are initialized to zero (0).

Typing

Normally, variables are typed by default or with the following explicit type declaration statements: DECIMAL, INTEGER, or REAL. (The DEFINT, DEFSNG, and DEFDBL statements are also provided for compatibility with other BASIC products but are not recommended for new programs.) The explicit type declaration 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; also defines the variable INT1 as type integer.

The rightmost (suffix) character of a numeric variable name may also be used to designate its type. (The type designator suffix is provided for compatibility with other BASIC products.)

A variable name ending with the character % is assigned integer type.

A variable name ending with the character # is assigned:

1. Decimal type, if the ARITHMETIC DECIMAL option is in effect
2. Real double type, if the ARITHMETIC NATIVE option is in effect

If a variable name does not end with # or % and the name or first letter of the name is not specified in an explicit typing statement, the variable is assigned:

- Decimal type, if the ARITHMETIC DECIMAL option is in effect
- Real single type, if the ARITHMETIC NATIVE and SPREC options are in effect
- Real double type, if the ARITHMETIC NATIVE and LPREC options are in effect

See "OPTION Statement" on page 312 for details on options.

Variable names ending with # or % may not be assigned a contradictory type. For example, DECIMAL X% is an error, but INTEGER X% is acceptable. The # and % suffixes override first character typing (with explicit typing statements).

Example

```
100 OPTION ARITHMETIC DECIMAL
110 INTEGER(A-C)
120 ALPHA#=BETA
```

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

Identifiers with self-typing characters (% or #) are only allowed in type declarations where the type for the self-typing character agrees with the type for the declaration. Note that such an identifier with a # self-typing character may become valid or invalid in a typing declaration by changing the ARITHMETIC or SPREC/LPREC options.

Example

```
100 REAL A#
```

is only valid if the ARITHMETIC NATIVE and LPREC options are in effect.

Example

```
200 DECIMAL A#
```

is only valid if the ARITHMETIC DECIMAL option is in effect.

Example

```
300 REAL DOUBLE A#
```

is valid only if the ARITHMETIC NATIVE option is in effect.

Example

```
400 DECIMAL A%
```

is always an error.

Figure 6 on page 34 summarizes the typing rules.

	ARITHMETIC DECIMAL		ARITHMETIC NATIVE	
	SPREC	LPREC	SPREC	LPREC
default	decimal	decimal	single	double
INTEGER/DEFINT	integer	integer	integer	integer
%	integer	integer	integer	integer
REAL SINGLE	single	single	single	single
REAL DOUBLE	double	double	double	double
REAL	single	double	single	double
DEFSNG	decimal	decimal	single	single
DEFDBL	decimal	decimal	double	double
DECIMAL	decimal	decimal	decimal	decimal
#	decimal	decimal	double	double

Figure 6. Typing Rules for Numeric Data

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. If the ARITHMETIC DECIMAL option is in effect and A is typed decimal (either by a DECIMAL statement or by default), then A and A# refer to the same variable. If A is typed integer, A and A% refer to the same variable. If the ARITHMETIC NATIVE option is in effect and A is typed real double, then A and A# refer to the same variable.

Since there is no implicit typing character for real single, if A is typed as real single it will not be the same as any other variable. A# could be decimal, A% integer, and A real single, all in the same program unit.

Automatic Initialization

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.

Internal Representation

Integer variables contain 32-bit twos-complement values as described under "Internal Representation of Integer Data" on page 27.

Decimal variables contain 17-digit values with a decimal point and power of 10 exponent. The internal representation and range of decimal values are discussed under "Internal Representation of Decimal Data" on page 28.

Real single variables are stored in floating-point hexadecimal format and occupy a full word (32 bits) of storage. For more information see "Internal Representation of Real Single Data" on page 29.

Real double variables are also stored in floating-point hexadecimal format and occupy a double word of storage. They are discussed under "Internal Representation of Real Double Data" on page 30.

Character Variables

Character variables:

- Must have names ending with $
- Contain strings which are variable in length
- Have a maximum length which is either:
  1. Declared explicitly in a DIM, COMMON or DEF statement
  2. The IBM-distributed default of 18 or a different default if set by your system administrator
- Have a maximum length which may not exceed 32767 characters
- Are initialized to a null (zero length) string

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 and a maximum length. The current length is the number of characters contained in the character value presently assigned to the variable. The maximum length is the maximum number of characters which can be assigned to the variable. The maximum length may be declared with the DIM, COMMON, or DEF statements or, if not declared, defaults to a value determined by your system administrator (18 is the IBM-distributed default). The maximum value that may be declared is 32767.

Each character of a character variable may have any hexadecimal value between 00 and FF, inclusive. However, character strings which contain control characters and other non-printable characters may cause adverse effects if they are output to files or terminals. On output to 327x type terminals in scroll mode, control characters (01 through 3F) are converted to spaces.

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.

Automatic Initialization

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

Internal Representation

Character variables are stored as described under "Character Data Type" on page 31. For a character variable, *max* is the maximum length defined for the variable and *cur* is the current length of the value associated with the variable. (See Figure 5 on page 31.)

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, integer, real single, real double, or decimal) and, for 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*, or linear sequence, 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. To do so, you must provide a subscript for each dimension of the array.

A subscript can be any numeric constant, numeric variable, or numeric expression whose rounded integer value is equal to or greater than zero. A set of subscripts is specified within parentheses after the array name to which they refer. Subscripts within the parentheses are separated by commas.

Example

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

The subscript reference (3,3,3) identifies a specific element within the three-dimensional array named ARA$.

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 value in ARA1$, corresponding to the subscript value 2, to the variable VAR$, you can use the statement:

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

If you define ARA2$ in this manner:

```
100 DIM ARA2$(2,3)
```

a two-dimensional array is generated, and you can refer to any one of the elements within ARA2$ by using the proper pair of subscripts.

MAT Statement Reference to Array Elements

In MAT statements, the elements of an array are accessed with the rightmost subscripts varying most rapidly. A MAT statement operating on ARA2$ above would refer to the elements of ARA2$ as follows. (Here we assume index values begin at 1, not 0. See "Base Option.")

```
ARA2$(1,1)
ARA2$(1,2)
ARA2$(1,3)
ARA2$(2,1)
ARA2$(2,2)
ARA2$(2,3)
```

Subscript Boundaries

A subscript boundary is the smallest or largest number which may be used as a subscript to refer to an element of a particular array.

The lower boundary of each subscript is determined by the BASE option in effect.

- If BASE 1 is in effect, the lower boundary of a subscript is one (1).
- If BASE 0 is in effect, the lower boundary of a subscript is zero (0).

Upper subscript boundaries are dependent on 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 39 and "Implicit Dimensioning of Arrays" on page 40), but may be dynamically changed by redimensioning (see "Redimensioning" on page 41). The intrinsic function SIZE may be used to determine the number of elements in a given dimension. The intrinsic function UDIM may be used to determine the maximum subscript value of a given dimension.

If a subscript value is outside 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 298).

Base Option

Array dimensioning depends on whether the subscript values begin at 0 or 1. Before dimensioning your arrays, you can specify either of these starting values in your program using the OPTION statement.

If BASE 0 is specified, the array is referenced starting with zero: A(0), A(1), A(2), A(3), and so forth. If BASE 1 is specified, the reference starts with one: A(1), A(2), A(3), A(4), and so forth.

Figure 7 shows how the size and subscripts of a one-dimensional array are affected by the base of the subscript.

```
100 OPTION BASE 0                100 OPTION BASE 1
110 DIM A(2)                     110 DIM B(2)

A contains three elements:       B contains two elements:

A(0)                             B(1)
A(1)                             B(2)
A(2)
```

Figure 7. One-Dimensional Array Elements—BASE 0 and 1

Figure 8 shows how the size and subscripts of a three-dimensional array are affected by the base of the subscripts.

```
Program Statements               Program Statements

100 OPTION BASE 0                100 OPTION BASE 1
110 DIM C(2,2,2)                 110 DIM D(2,2,2)

C contains 27 elements:          D contains 8 elements:

C(0,0,0)                         D(1,1,1)
C(0,0,1)                         D(1,1,2)
C(0,0,2)                         D(1,2,1)
C(0,1,0)                         D(1,2,2)
C(0,1,1)                         D(2,1,1)
C(0,1,2)                         D(2,1,2)
C(0,2,0)                         D(2,2,1)
C(0,2,1)                         D(2,2,2)
C(0,2,2)
C(1,0,0)
C(1,0,1)
.
.
.
C(2,2,0)
C(2,2,1)
C(2,2,2)
```

Figure 8. Three-Dimensional Array Elements—BASE 0 and 1

Note: A series of values assigned to a multidimensional array (for example, with a MAT READ statement) 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 statement.

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:

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

dimensions a numeric array named ARRAY to 20 rows and 4 columns, for a total of 80 elements.

The upper bound for a dimension must be an unsigned integer constant in the range 1 to 32767 for BASE 1 or 0 to 32767 for BASE 0. Otherwise, the total maximum size of an array is limited only by the available virtual storage. See "DIM Statement" on page 188 and "COMMON Statement" on page 170.

Numeric Arrays

Numeric arrays are typed (integer, decimal, real single, and real double) in the same manner as numeric variables (see "Numeric Variables" on page 32):

- The explicit type declaration statements can declare specific array names or specific beginning letters of array names.
- The characters # and % at the end of array names override initial character declarations in explicit type declaration 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.

Elements of numeric arrays contain values as described for numeric variables and are initialized to zero. (See "Numeric Data Types" on page 26.)

Character Arrays

Character arrays follow almost the same rules as character variables (see "Character Variables" on page 35).

- 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-distributed default.)
- Each element of a character array has a current length which is not necessarily equal to the current length of other elements. Character arrays contain values as described for character variables (see "Character Data Type" on page 31).

- 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.

Implicit Dimensioning of Arrays

If the first usage of a name in a program unit 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 whether the BASE 0 or BASE 1 option is in effect.

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

For example, if the reference has three subscripts in an assignment statement, three dimensions are assumed.

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.

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

For example, assuming the BASE 0 option is in effect, there is no explicit dimensioning, and this is the first reference:

- MAT (array assignment) statement

  ```
  50 MAT A=(15)
  ```

  establishes A as a two-dimensional array, each dimension containing 11 elements (subscript values 0 through 10). All 121 elements receive the value 15.

- MAT name in an input or output list

  ```
  70 PRINT USING 120:MAT ARAY
  ```

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

- The argument of an intrinsic function which requires an array parameter

  ```
  80 IF DOT(A,B)=0 THEN 200
  ```

  establishes A and B as one-dimensional arrays each containing 11 elements.

- Reference with subscripts

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

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

- Reference with subscripts

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

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

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

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

Redimensioning

After 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 provided the number of elements in the original array is not exceeded.

Redimensioning is done as follows:

1. Array size is verified. If the number of elements in the array as redimensioned exceeds the original number of elements in the array, an exception occurs.

2. If there is sufficient space to allow redimensioning, the view of storage is changed as if there were an element by element transfer from the array as it was before redimensioning, into the shape that it assumes. The elements are transferred in the order by which they are stored (the rightmost subscript varying most rapidly). If there are insufficient items in the current array to fill the new array, the new data items are filled with zeros for numeric arrays or set to null for character arrays.

Example

Original Dimensions	**Redimensioning Reference**
`OPTION BASE 1`	
`DIM ARRAY1(100)`	`MAT ARRAY1 = ARRAY1(85)`
`DIM ARRAY1(100)`	`MAT ARRAY1 = ARRAY1(10,10)`
`DIM ARRAY2(20,20)`	`MAT ARRAY2 = ARRAY2(300)`

The following redimensioning of ARRAY2 is invalid; the redimensioned ARRAY2 would exceed the original size of ARRAY2:

Original Dimensions	Redimensioning Reference
`DIM ARRAY2(20,20)`	`MAT ARRAY2 = ARRAY2(500)`

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

Example

```
Main Program                        Subprogram

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

After control returns to the main program, ARRAY1 has the dimensioning ARRAY1(5,5,4).

Functions

IBM BASIC supports three categories of functions; user-defined functions, intrinsic (scalar) functions, and array functions. User-defined functions are defined within your program while intrinsic functions and array functions are built into BASIC, and are available as part of the language.

User-defined and intrinsic functions compute a single value and are invoked by reference (that is, by using the function name in a statement where a value is required). Invocation is typically done:

- In an expression
- To the right of the equal sign in a LET statement
- In the output list of a PRINT statement.

The name of a user-defined or intrinsic function indicates whether the function returns a numeric or character value:

- Function names not ending in a dollar sign return numeric values.
- Function names ending in a dollar sign return character values.

Array functions are unique in that they return only array results, and may be invoked only within the MAT (array assignment) statement. Many of these functions can return either character or numeric results, depending on the type of their argument.

Functions can accept values from the invoking statement through the use of a list of arguments in parentheses following the function name. If the argument list consists of more than one argument, commas separate them. Valid types of arguments (character or numeric) and valid argument ranges for each intrinsic or array function are predefined. With user-defined functions, the valid argument types and value ranges are specified by the user as part of the function definition.

Examples of function invocation

```
120 LET area = PI * radius**2          ! Pi function has no arguments
160 PRINT SIN(radians)                 ! Sine function     - 1 argument
220 LET a$ = LPAD$("Salary", 10)       ! Left pad function - 2 arguments
290 IF INT(100*RND+1) = 13 THEN STOP   ! RND function nested within INT
340 MAT array_b$ = TRN (array_a$)      ! Transpose array function
```

When the result of a function is displayed with a PRINT statement, the number of digits printed will depend on the print options in effect, and may therefore differ from the number of digits actually returned by the function.

The following definitions apply to the contents of the parentheses (if parentheses are present), following the function name:

parameter One in a list of one or more identifiers separated by commas:

- In a DEF statement for a user-defined function or
- As specified by the description of an array or intrinsic function.

The identifiers represent BASIC variables or arrays and are local to the function.

argument One in a list of one or more items separated by commas in an executable statement in which the value of the function must be evaluated. The item may be:

- A constant
- A variable name
- An expression
- A user-defined or intrinsic function reference
- An array name for certain intrinsic and array functions.

The value of a numeric argument will be converted to the same type as the corresponding parameter, which must also be numeric. A character argument must correspond to a character parameter.

Note that parameters are specified in the *definition* of a function, while arguments are specified in the *invocation* of a function. Parameters are sometimes known as "formal parameters" or "dummy variables," while arguments are sometimes known as "actual parameters."

User-Defined Functions

You can define and name your own (user-defined) functions within your program unit. See the "User-Defined Function Statements" on page 90, the "DEF Statement" on page 180 and the "FNEND Statement" on page 202 for more information.

Intrinsic Functions

An intrinsic function is a predefined function supplied by BASIC to perform common operations and return frequently used numeric and character values. For a detailed explanation of each intrinsic function, and examples of how they are used, see "Intrinsic Functions and Array Functions" on page 379.

Intrinsic Functions Using Double Precision

The following intrinsic functions convert integer, decimal, and real single arguments to real double, and compute the result using double precision binary floating-point operations. When the argument (or both arguments) are integer, the result type depends on the options in effect (see "When the Result Type Depends on the Options in Effect" on page 46). Otherwise, the double precision result type of these functions is the same as the type of the argument (or the same as the

argument with the highest type; the priority of numeric types, from highest to lowest, is decimal, real double, real single, and integer).

ACOS(X)	COT(X)	SEC(X)
ANGLE(X,Y)	CSC(X)	SIN(X)
ASIN(X)	EXP(X)	SINH(X)
ATN(X)	LOG(X)	SQR(X)
COS(X)	LOG2(X)	TAN(X)
COSH(X)	LOG10(X)	TANH(X)

Special Cases of Numeric Result Type

Six of the intrinsic functions: DET (without an argument), EPS, INF, PI, RND (without an argument), and VAL(C$), have result types that vary, depending on the context in which they are used. There are three special cases:

1. When a function is used as part of an expression.
2. When a function is used in a LET statement.
3. When a function is used as an argument of a user-defined function.

In the first case: The result type of the function depends on the type of the operand it is combined with. If the other operand is decimal or real, the function's result type is the same as that of the other operand. If the operand it is combined with is integer type, or when the type of the operand has not been determined (for example, when that operand is a constant), or when there is no other operand, the function's result type is determined by the options in effect (see "When the Result Type Depends on the Options in Effect" on page 46). If ARITHMETIC DECIMAL is in effect, the result is decimal, and if ARITHMETIC NATIVE is in effect, the result is real (real single if SPREC is in effect, and real double if LPREC is in effect).

In the second case:

- If the LET statement assigns the value of the function to one or more variables that are either all decimal type, or all real type, the result type of the function is the same as the type of the variable(s).
- If the variables assigned in the LET statement have different types, the result type of the function is the same as the type of the variable with the highest type; priority of numeric types, from highest to lowest, is decimal, real double, real single, and integer).
- If all the variables assigned in the LET statement have integer type, then the result type of the function depends on the options in effect, just as in the first case.

In the third case:

- If the corresponding parameter is integer, the result type of the function depends on the options in effect, just as in the first case.
- If the corresponding parameter does not have integer type, the result type is the same as the parameter type.

When the Result Type Depends on the Options in Effect

In most cases, the result type of an intrinsic function is determined by the specific function, its context, and the type of its arguments. This is not true, however, for the functions listed in "Intrinsic Functions Using Double Precision" on page 44, the functions CEN, DEG, DET (with an argument), FAH, RAD, RND (with an argument), ROUND, and TRUNCATE. For these functions, if the first argument is integer (or for the ANGLE function, if both arguments are integer), then the ARITHMETIC DECIMAL | NATIVE and SPREC | LPREC options determine the function's result type. These cases are noted in each function description.

For these cases, the result type is:

- Decimal, if ARITHMETIC DECIMAL is in effect.
- Real single, if ARITHMETIC NATIVE and SPREC are in effect.
- Real double, if ARITHMETIC NATIVE and LPREC are in effect.

The result of the REAL function is:

- Real single, if SPREC is in effect.
- Real double, if LPREC is in effect.

(See "OPTION Statement" on page 312).

Array Functions

Array functions are also predefined and supplied with BASIC. They differ from intrinsic functions in the following ways:

- They return an array result.
- Many array functions can return either a character or numeric array result, depending on the type of their argument.
- They can be invoked only within the MAT assignment statement.

For a detailed explanation of each array function see "Array Functions" on page 286.

Expressions

Expressions are representations of numeric or character values. An expression can be any one of the following:

Numeric	Numeric values (constants and/or variables and/or numeric functions), optionally combined by numeric operators.
Character	Character values (constants and/or variables and/or character functions), optionally combined by character string operators.
Relational	Numeric expressions combined by relational operators or character expressions combined by relational operators.
Logical	Relational expressions combined by logical operators.
Array	Entire numeric or character arrays, and/or functions which yield a numeric or character array result, 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 called scalar operators, as shown in Figure 9 on page 48.

Evaluation of Numeric Expressions

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 9 on page 48.

Operator	Meaning	Evaluation Order
** or ¬ or ^	Exponentiation	1
*	Multiplication	2
/	Division	2
+	Addition (or sign operator)	3
-	Subtraction (or sign operator)	3

Figure 9. Numeric Operators and Evaluation Order

Note: One or more spaces may appear between the two asterisks of the exponentiation operator.

The left to right processing of operators is modified to provide compatibility with accepted practices of arithmetic processing. The highest processing priority is given to exponentiation. The intermediate processing priority is given to multiplication and division. If these two operators are 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 in the same expression, normal left to right processing priority is used. See Figure 9.

Evaluation of the following expression follows left to right processing, because the operators are of equal order:

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 expression 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 remaining 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.

Note: Actual evaluation of an expression may be changed where permitted by the associative and commutative rules of arithmetic. This may cause underflow or overflow exceptions to occur (or not to occur) which would otherwise not occur (or occur) if the above rules were strictly followed. Loss or increase in accuracy may also occur due to reordering. You can use parentheses to force a desired order of evaluation.

Parentheses in Numeric Expressions

Parentheses provide a means to change 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 most deeply 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. Lastly, 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 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; A*(B*C) does not necessarily give the same results as (A*B)*C.

This is the case when an intermediate computation results in a value that is larger or smaller than machine capacity. This is known as overflow or underflow.

For example (assuming the ARITHMETIC DECIMAL option is in effect):

```
1E600*(1E500*1E-500) equals 1E600
```

but

```
(1E600*1E500)*1E-500 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

```
0.9999999999999999E+999 (decimal infinity)
```

to be substituted into the expression that is then multiplied by 1E-500. This results in the total expression equal to:

```
0.9999999999999999E+499
```

Plus and Minus as Sign Operators

The plus (+) and minus (-) signs can also be used as positive and negative signs, 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

Examples

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

Mixed Type Numeric Expressions

The data type of a numeric expression depends on the data types of its operands. If all the operands in the expression have the same type, whether it is integer, decimal, real single, or real double, the result of the expression has the same type as the operands (with the exception of integer division and exponentiation, which are described below).

When a numeric operator (+, -, *, /, or **) acts on two operands with different types, one of those operands must be converted to match the other before BASIC can perform the operation. The operand of lower type priority is usually converted to match the operand of higher type. There is one case in which no conversion takes place; when a decimal or real operand is raised to an integer power, the integer is not converted, but the result type is the same as the type of the base.

Priority of numeric types, from highest to lowest, is decimal, real double, real single, and integer. Figure 10 shows which operand is converted for every case in which operand types differ.

	INTEGER	SINGLE	DOUBLE	DECIMAL
INTEGER	*	single	double	decimal
SINGLE	single	single	double	decimal
DOUBLE	double	double	double	decimal
DECIMAL	decimal	decimal	decimal	decimal

Figure 10. Result Type of Expressions

* When the operator is + , -, or *, the result type is integer. For / and **, see "Integer Division" on page 52 and "Integer Exponentiation" on page 52.

Conversion of an operand is the same as if the following intrinsic functions had been applied to the operand:

Conversion to real single— SNG
Conversion to real double— DBL
Conversion to decimal— DEC

Note: Precision may be lost if an integer operand is converted to real single (integer has up to 9 digits of precision while real single has only 6 digits of precision).

The result type of a numeric expression with one or two operands is independent of the expression's context. For instance, if the expression with one or two operands is a part of a larger expression, none of the components of that larger expression have an effect on the result type of the smaller expression. For example, if X is decimal type, J is integer type, and IFIX is the decimal to integer conversion function, the expression IFIX(X+J) is an integer expression and X+J is a decimal expression.

Decimal exponentiation to a noninteger power is computed using real double; therefore the precision of the result is (approximately) 15 decimal digits instead of 17.

Integer Division

One operand of type integer may be divided by another operand of type integer. Since the mathematical quotient of two integers is not necessarily an integer, both operands are converted to decimal or real prior to the division and the result is of the same type.

The conversion and result type are decimal if ARITHMETIC DECIMAL is in effect, real single if ARITHMETIC NATIVE and SPREC are in effect, and real double if ARITHMETIC NATIVE and LPREC are in effect (see "OPTION Statement" on page 312).

Note: Precision may be lost if the integer operands are converted to real single (integer has up to 9 digits of precision while real single has only 6 digits of precision).

Integer Exponentiation

Integer exponentiation follows the same rules as integer division. If, in the expression (J1**J2), J2 is negative, BASIC interprets the expression as 1/(J1**ABS(J2)). This expression is subject to the rules for integer division. Because BASIC doesn't know whether J2 will be negative, all cases of integer exponentiation, even those where J2 is positive, are treated like integer division.

BASIC converts J1 to decimal or real, and then performs the exponentiation. The conversion and result type of decimal, real single, or real double are determined as for integer division.

Character Expressions

Character expressions are composed of combinations of character constants, character variables, character array elements, and character-valued function references, combined by the concatenation operator and modified by substring qualifiers.

Examples

```
"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)        (reference to a character function)
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"). 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 Expressions

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 using substring qualifiers.

A substring of a character string is specified by adding a substring qualifier to a character constant, character variable, reference to a character function, reference to a character expression (enclosed in parentheses), or character array element.

A substring qualifier has the form:

```
(m:n)
```

where:

m is the beginning position of the substring.
n is the ending position of the substring.

The positions in a character string are numbered starting with one. 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.

The following statements illustrate valid substrings:

```
10  P$ = "ABCDE"(2:3)                        !P$ = "BC"
20  A$ = "BC": B$ = "EFGHIJKLMN"
30  Q$ =  A$&B$(4:7)                         !Q$ = "BCHIJK"
40  R$ = (A$&B$)(4:7)                        !R$ = "FGHI"
50  S$= ((A$&B$)(4:7))(2:3)                  !S$ = "GH"
60  DEF USER_DEF$(X$,Y$) = X$&Y$
70  T$ = USER_DEF$('123', '456')(3:4)        !T$ = "34"
```

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

Substrings are treated as follows:

- 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$).
- If *m* is greater than *n*, the addressed substring is the null string preceding the *m*th character of A$.

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$(2:3)     Assigns "WX" (the second and third
                 positions of B$) to G$

G$ = B$(4:4)     Assigns "Y" (the fourth position of B$)
                 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"        Replaces "CD" in A$ with "PQ",
                       giving "ABPQEF"

A$ (3:4) = ""          Deletes "CD" from A$,
                       giving "ABEF"

A$ (3:4) = B$(1:2)     Replaces "CD" in A$ with "VW" from B$,
                       giving "ABVWEF"

A$ (3:4) = B$(3:3)     Replaces "CD" in A$ with "X from B$,
                       giving "ABXEF"

A$ (3:4) = B$(1:4)     Replaces "CD" in A$ with "VWXY" from B$,
                       giving "ABVWXYEF"

A$ (3:2) = B$(1:4)     Inserts "VWXY" from B$ before "C" in A$,
                       giving "ABVWXYCDEF"
```

Preceding Insertion

```
A$(1:0) = "PQ"        Inserts "PQ" before "A" in A$,
                      giving "PQABCDEF"

A$(1:0) = B$(3:4)     Inserts "XY" from B$ before "A" in A$,
                      giving "XYABCDEF"

A$(1:0) = B$(1:1)     Inserts "V" from B$ before "A" in A$,
                      giving "VABCDEF"
```

Trailing Insertion (Appending)

```
A$(7:8) = "PQ"        Inserts "PQ" after "F" in A$,
                      giving "ABCDEFPQ"

A$(7:8) = B$(2:4)     Inserts "WXY" from B$ after "F" in A$,
                      giving "ABCDEFWXY"

A$(10:11) = "PQ"      Inserts "PQ" after "F" in A$,
                      giving "ABCDEFPQ"
```

Substring qualifiers can also be used with 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.

Alternately, this could be done with the statement:

```
130 PRINT DATE$(4:8) & DATE$(3:3) & DATE$(1:2)
```

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 CASE (in abbreviated form), DO, EXIT IF, IF, and LOOP statements.

Relational Operators

Relational operators are defined as shown in Figure 11.

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 11. Relational Operators

Note: One or more spaces may appear between the two characters in the relational operators <>, ><, >=, =>, <=, and =<.

Numeric Data in Relational Expressions

Mixed type numeric relational expressions are allowed. When the operands of a numeric relation have different types, the operand with lower type is converted to the type of the other operand before the comparison is made. Decimal type is highest priority, then real double, then real single, then integer. (See also "Mixed Type Numeric Expressions" on page 51).

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 option on the OPTION statement.

When COLLATE NATIVE is specified, the EBCDIC collating sequence is used. When COLLATE STANDARD is specified, the ASCII collating sequence is used. Both collating sequences are listed in Appendix C, "Character Set Collating Sequences" on page 505.

If neither option is specified, the default is used. See "SET OPTION Command" on page 483 for how to change the defaults. The IBM-distributed default is COLLATE NATIVE.

Figure 12 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 12. 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 the longer expression in value. For example, "ABC" is less than both "ABCD" and "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 DO, EXIT IF, IF, and LOOP statements.

Note: Not all of the expressions in a logical expression need be evaluated if the value of the logical expression can be determined without evaluating all of them. Conversely, the value of an expression in a logical expression may be evaluated even if its value is not necessary to determine the value of the logical expression.

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 at least one 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 NOT must precede a logical expression.

Combining Logical Expressions

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

Example

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

is a valid logical expression. The parentheses around

```
NUM<=COUNT OR DATA$ = "END"
```

cause this expression to be evaluated first and are required if your intention is to negate the result of this expression.

Priority of Expression Evaluation

The priority of evaluation of expressions is shown in Figure 13 on page 59.

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

Figure 13. Scalar Expressions—Evaluation Priority

Parentheses provide a means of modifying the evaluation order shown in Figure 13. 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.

The following program segment uses all the rules of priority of expression evaluation:

Example

```
400   LET N = 1
410   LET A$ = 'YES'
420   IF N=1 OR NOT 2+3=5 AND A$='YES' THEN PRINT 'TRUE'
```

The IF statement at line 420 evaluates the logical expression between the IF and the THEN in the chronological sequence shown below:

```
     Evaluation Step              Priority    Comments

N=1  OR NOT 2+3=5 AND A$='YES'              Original expression
N=1  OR NOT  5 =5 AND A$='YES'    1         Evaluate 2+3
TRUE OR NOT  5 =5 AND A$='YES'    2         Evaluate each relational
TRUE OR NOT  TRUE AND A$='YES'    2            expression from left to
TRUE OR NOT  TRUE AND  TRUE       2               right
TRUE OR   FALSE   AND  TRUE       3         Evaluate NOT TRUE
TRUE OR          FALSE            4         Evaluate FALSE AND TRUE
    TRUE                          5         Evaluate TRUE OR FALSE
```

Therefore, the IF statement executes the PRINT, producing the output TRUE.

Parentheses can be used for clarity even when they do not override existing priorities. The expression:

```
(N = 1) OR (NOT (2+3=5) AND (A$='YES'))
```

produces the same result as above, yet is easier to understand.

Note: Actual evaluation of a logical expression may be changed where permitted by commutative and associative rules of logical operators. This may cause underflow or overflow exceptions to occur (or not to occur) which would otherwise not occur (or occur) if the above rules were strictly followed. Parentheses may be used to force a desired order of evaluation.

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.

Mixed type numeric array operations are allowed. For example, an integer array may be added to a decimal array, or a decimal scalar value may be multiplied by a real array. The resulting type will be the highest type of the operands, as for scalar operations. then integer (see "Mixed Type Numeric Expressions" on page 51).

Array expressions may appear only within the MAT statement. See "MAT (Array Assignment) Statement" on page 276 for a description of array expressions.

The intrinsic functions DET, DOT, PRD, SRCH, and SUM also perform operations on entire arrays.

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.

For a more detailed discussion and examples of how different files are used, see *IBM BASIC Programming Guide.*

File handling, naming, and structure are system dependent. See your IBM BASIC System Services manual for details and restrictions.

File Attributes

All files have attributes which describe how the data is formatted, how the data is organized, and how the data can be accessed. These four attributes:

```
records
type
organization
access
```

interact according to the rules specified in this section. (The POINTER attribute is discussed in the "OPEN Statement" on page 305).

File Records

The collection of data values of which a file is composed 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 items, 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 (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 provides a fixed number of positions whether the employee has one dependent or 20.

Variable record length files are those whose records do not necessarily have the same length; 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 (see "OPEN Statement" on page 305).

File Type

The file type attribute describes the type of records in a file. The TYPE clause of the OPEN statement permits specification of the file type.

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

Display Type

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

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

DISPLAY files are easy to edit. You may format them with FORM or IMAGE or leave them unformatted.

Internal Type

INTERNAL file type 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 character, integer, decimal, or real). Consequently, files of this type cannot be edited easily.

Native Type

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

File Organization

The file organization attribute describes how data is arranged in a file. The way a file is organized determines how it can be accessed.

The file organization can be SEQUENTIAL, STREAM, RELATIVE, or KEYED, and is specified in the ORGANIZATION clause of the OPEN statement.

Sequential Organization

In a SEQUENTIAL file, records are stored in the same order that you put them in. The first record occupies the first position in the file, and 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

STREAM organization is just like sequential organization, except that each record consists of a single value in internal binary format (see "Internal Type" on page 62).

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 reading is sequential, null (deleted) records are automatically bypassed.

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

Keyed Organization

A file with keyed organization consists of records identified *by key*. A key is a string of characters contained at a specific location within the record. The location and length of the key are the same for all records in the file.

In a keyed file, you can access any record *directly* by using the record key, or you can access all of them *sequentially* (in the order the keys collate). File positioning can be made by reference to a portion of the key. Keyed organization requires that VSAM be used. See "Related Publications" on page iv for the VSAM documentation appropriate for your system.

File Access

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

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

INPUT Access

INPUT File access 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

OUTPUT file access 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

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 which permits rewriting or deletion of records. Sequential files may be extended by adding more records, but they cannot be shortened.

Combinations of File Type, Organization, Access, and Records

Not all combinations of file type, organization, access, and records are permitted. The allowed combinations are in Figure 14.

Type	Organization	Access	Records
DISPLAY	SEQUENTIAL	INPUT [1] OUTPUT OUTIN [1]	FIXED VARIABLE
INTERNAL	STREAM	INPUT OUTPUT	FIXED VARIABLE
INTERNAL	SEQUENTIAL	INPUT OUTPUT OUTIN	FIXED VARIABLE
NATIVE	SEQUENTIAL	INPUT OUTPUT OUTIN	FIXED VARIABLE

Figure 14 (Part 1 of 2). Valid File Attribute Combinations

Type	Organization	Access	Records
NATIVE	RELATIVE	INPUT OUTPUT OUTIN	FIXED
NATIVE	KEYED	INPUT OUTPUT OUTIN	FIXED VARIABLE

Figure 14 (Part 2 of 2). Valid File Attribute Combinations

1 DISPLAY files cannot be opened with INPUT or OUTIN access if "DEVICE PRINTER" or "DEVICE 3800" is specified.

File Input/Output Statements

File input/output statements process data which is stored in files. 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. See also "Input/Output Statements" on page 84.

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 allowed.

In addition to OPEN, several file input/output statements allow for a "positional" clause which specifies a particular position.

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 relational condition. The KEY condition is applied to the full length of each 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 relational condition. The SEARCH condition can specify a partial key (a key whose length is less than the full length of the file's key).

The input/output statements which allow record positioning are shown in Figure 15 on page 66.

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

Figure 15. Positioning Options Allowed—File Input/Output Statements

File Control Statements

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

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

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

RESET #fileref Changes the position of the file pointer for a file. The RESTORE keyword may be used instead of RESET.

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.

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

DELETE #fileref Removes a specific record from a (native) keyed or relative file.

GET #fileref Assigns values from an internal (stream or sequential) type file to a list of data items.

INPUT #fileref	Assigns values from files in three different ways. For display files, it functions as an INPUT statement; for sequential internal files, it functions as a READ #fileref statement; and for internal stream files, it functions as a GET #fileref statement.
LINE INPUT #fileref	Assigns a complete record of unformatted data to a character variable from a display file.
PRINT #fileref	Transmits both formatted and unformatted data to a display file. Values are formatted with FORM or IMAGE.
PUT #fileref	Writes values from a list of data items to a stream file.
READ #fileref	Retrieves a record from an internal sequential 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.
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.
WRITE #fileref	Adds a record to a native or internal (stream or sequential) 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.

Exception Recovery Clauses

You can provide exception recovery from some file I/O statements by using the exception recovery clauses. For more information on exception recovery clauses, see "Using I/O Statement Exception-Recovery Clauses and On Condition Statements" on page 127, or the individual clause.

Not all exception recovery clauses can be used with all file I/O statements, however. Figure 16 on page 68 shows which clauses can be used with which statements.

	CONV	DUPKEY	DUPREC	ENDPAGE	EOF	IOERR	NOKEY	NOREC	SOFLOW	EXIT
CLOSE					X	X				X
DELETE						X	X	X		X
GET	X				X	X			X	X
INPUT	X					X			X	X
INPUT FIELDS	X					X			X	X
INPUT File	X				X	X			X	X
LINPUT						X			X	X
LINPUT File					X	X			X	X
MARGIN										
MARGIN File										
OPEN						X				X
PRINT	X					X			X	X
PRINT FIELDS	X					X			X	X
PRINT File	X			X	X	X			X	X
PUT					X	X				X
READ	X				X				X	X
READ File	X				X	X	X	X	X	X
REREAD	X					X			X	X
RESET						X	X	X		X
REWRITE	X				X	X	X	X	X	X
SCRATCH						X				X
WRITE	X	X	X		X	X			X	X

Figure 16. Exception Recovery Clauses for I/O statements

File Statements and File Attributes

Not all file input/output statements can be used with all kinds of files. Figure 17 on page 69 lists which statements can be used with each of the six legal combinations of type and organization. Figure 17 on page 69 also notes possible uses of the various combinations.

Type	Organization	Statements	Use
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	GET INPUT PUT RESET BEGIN RESET END SCRATCH WRITE	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.
INTERNAL	SEQUENTIAL	INPUT READ RESET BEGIN RESET END SCRATCH WRITE	The only difference between a SEQUENTIAL, NATIVE and a SEQUENTIAL, INTERNAL file is the format of the records themselves.
NATIVE	SEQUENTIAL	READ USING REREAD USING RESET BEGIN RESET END REWRITE USING SCRATCH WRITE USING	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.
NATIVE	RELATIVE	DELETE REC READ USING REREAD USING RESET BEGIN RESET REC REWRITE USING SCRATCH WRITE USING	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.
NATIVE	KEYED	DELETE KEY READ USING REREAD USING RESET BEGIN RESET KEY REWRITE USING SCRATCH WRITE USING	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, and so forth. The file must be a VSAM file.

Figure 17. File Type, Organization, Statements, and Use

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, other than to continue with the next appropriate statement.

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
```

Although 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 the statements are listed alphabetically and discussed individually in "BASIC Statements" on page 145.

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

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

All the above categories of statements are discussed in the following sections, except for remark statements. Remark statements are discussed in "Comments" on page 15 and "REM Statement" on page 352.

Declarative Statements

Nine 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. (See "COMMON Statement" on page 170.)

DECIMAL The DECIMAL statement explicitly defines which identifiers in the program unit are to be assigned decimal type. (See "DECIMAL Statement" on page 178.)

DEFDBL The DEFDBL statement explicitly defines which identifiers in the program unit are to be assigned real double or decimal type. (See "DEFDBL Statement" on page 183.)

DEFINT The DEFINT statement explicitly defines which identifiers in the program unit are to be assigned integer type. (See "DEFINT Statement" on page 184.)

DEFSNG The DEFSNG statement explicitly defines which identifiers in the program unit are to be assigned real single or decimal type. (See "DEFSNG Statement" on page 185.)

Note: DEFDBL, DEFINT, and DEFSNG statements are provided for compatibility with other BASIC products and are not recommended for new programs. DECIMAL, INTEGER, or REAL statements should be used instead.

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. (See "DIM Statement" on page 188.)

INTEGER The INTEGER statement explicitly defines which identifiers in a program unit are to be assigned integer type. (See "INTEGER Statement" on page 258.)

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 the default and any specified in a RUN or COMPILE command. (See "OPTION

Statement" on page 312 and "SET OPTION Command" on page 483.)

REAL The REAL statement explicitly defines which identifiers in the program unit are to be assigned real type. (See "REAL Statement" on page 350.)

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 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 define 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 18. 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.

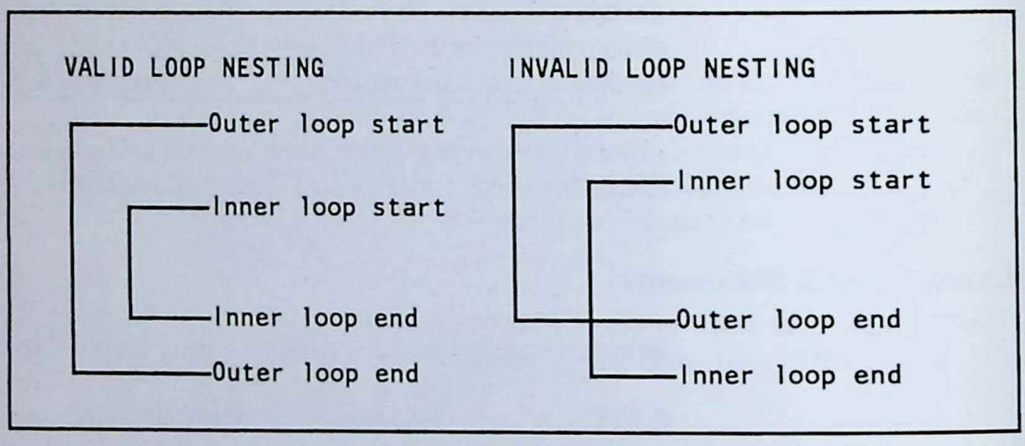


Figure 18. Valid and Invalid Loop Nesting

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.

See also "DO Statement" on page 190 for more information.

LOOP Indicates the last statement (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.

See also "LOOP Statement" on page 268 for more information.

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

Figure 19 shows the flow of control in a DO/LOOP block.

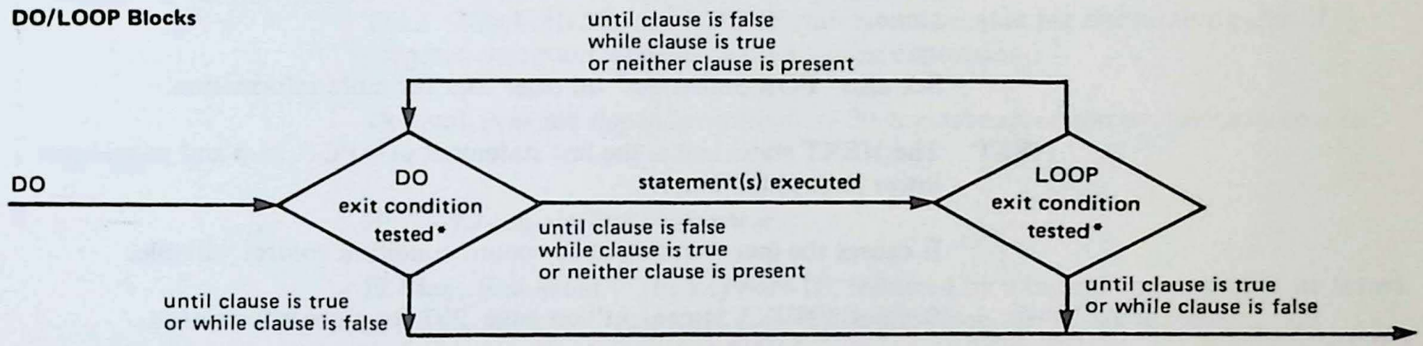


Figure 19. DO/LOOP Block Flow of Control

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, the WHILE or UNTIL clause 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, END SUB, and FNEND, each having been set originally by a condition (the conditions are GOSUB statements, CALL statements, exceptions, and function references) within that DO loop.

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

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 statement 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 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.

See also "FOR Statement" on page 203 for more information.

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

It causes the incrementing of the count condition control variable.

See also "NEXT Statement" on page 297 for more information.

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 20.

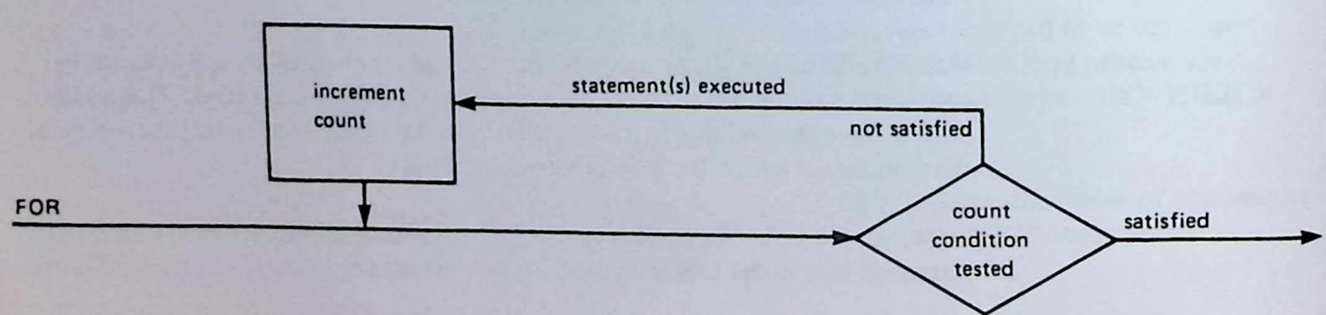


Figure 20. FOR/NEXT Loop Flow of Control

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, END SUB, and FNEND, each having been set originally by a condition (the conditions are GOSUB statements, CALL statements, exceptions, and function references) within that FOR loop.

A FOR loop may be exited by an EXIT IF statement (see "EXIT IF Statement" on page 200) and also by other branching statements.

Decision Structure Control Statements

These statements are tools for structured programming. They allow conditional processing of alternative statement sequences. The IF Block structure allows 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

The IF Block, ELSE, and END IF statements 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 five basic parts:

IF Block Statement The keyword IF, followed by a logical expression to be tested and the keyword THEN (see "IF Block Statement" on page 234).

THEN Block A block of statements surrounded by the IF Block statement and the ELSE statement (if present) or by the END IF statement if the ELSE statement is not present. This block is executed 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 Statement The marker that indicates the end of the THEN block and the beginning of the ELSE block. The ELSE statement is not required if there is no ELSE block (see "ELSE Statement" on page 192).

ELSE Block A block of statements surrounded by the ELSE statement and END IF statement.

The ELSE statement and the ELSE block form a logical pair. For the ELSE block to be recognized as the set of statements to be executed in the event the IF condition is *false*, the ELSE block must be preceded by the ELSE statement.

If an ELSE block is not present and the condition is *false*, no statements within the IF block are executed, and execution resumes following the END IF statement.

END IF Statement The last line of the IF block. Indicates the end of the ELSE block (if present), or the end of the THEN block if there is no ELSE block (see "END IF Statement" on page 194).

A simpler, but more restrictive, alternative to the IF Block is an IF statement (see "IF Statement" on page 231). The major differences between an IF statement and the IF block as described above (which uses an IF Block statement as part of the structure) are:

- An IF Block statement requires a matching END IF statement. The END IF statement is invalid for an IF statement.
- In the IF Block statement, the statements following THEN or ELSE may be coded on separate lines. In an IF statement, all the statements following both THEN or ELSE must be imperative statements and must be contained on a single line (or continuation line.)

Figure 21 shows this flow of control.

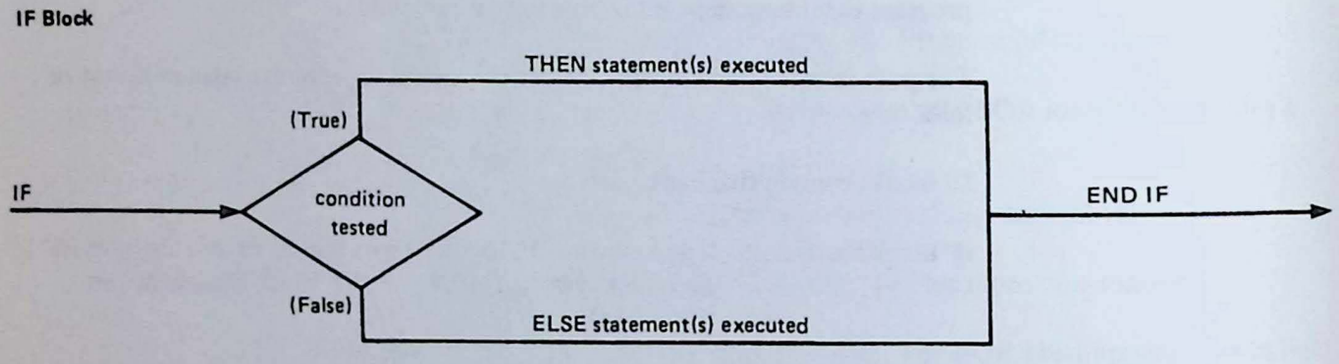


Figure 21. IF Blocks—Flow of Control

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
```

or, using multiple statements per line:

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

Here is the same process, but using an IF *statement* instead.

```
100 IF A=B THEN &
  &     LET C=10 : &
  &     LET E=20   &
  & ELSE           &
  &     LET E=10
```

Note: The indentation clarifies the IF block structure. You may indent any way you like.

The IF block in the example is executed as follows:

1. If A equals B, lines 110 and 120 are executed. Execution then skips to the line after END IF.

2. If A does not equal B, line 140 is executed. Execution 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, END SUB, and FNEND, each having been set originally by a condition (the conditions are GOSUB statements, CALL statements, exceptions, and function references) within that IF block.

An IF block can be exited:

- Normally, with the END IF statement
- By a branch control statement (GOTO, IF, ON)
- As the result of a call to a function, subroutine, or subprogram
- As the result of an exception

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 Statement

The SELECT statement is the initial statement of a SELECT block. It contains the select expression to be tested by the CASE statements that follow. See "SELECT Statement" on page 366.

CASE Block

There can be as many CASE blocks as are needed by the logic of the program. A CASE block is composed of:

CASE Statement

The CASE statement begins a CASE block and specifies the criteria for execution of its CASE block statements.

After the selection expression is evaluated, CASE statements are tested against it in the order of their

occurrence in the select block. The first CASE statement (if any) whose criteria are met by the evaluated selection expression has its CASE block executed. Execution then resumes after the END SELECT statement (any remaining CASE blocks are skipped).

See "CASE Statement" on page 162.

CASE Block Statements

A block of statements preceded by a CASE statement and followed by another CASE statement, a CASE ELSE statement, or the END SELECT statement. A block is executed under the conditions described in the CASE statement (above).

CASE ELSE Block (optional)

There can be at most one CASE ELSE block following the last CASE block.

If a CASE ELSE block is not present and no CASE block is executed, an ERROR exception occurs. (See "Exception Handling Statements" on page 127 and "ON Condition Statement" on page 298.)

If a CASE ELSE block is present, it consists of:

CASE ELSE Statement

The CASE ELSE statement begins the CASE ELSE block and immediately precedes the CASE ELSE block statements.

If no CASE block is executed, the CASE ELSE block is executed, and execution continues after the END SELECT statement.

See "CASE ELSE Statement" on page 164.

CASE ELSE Block Statements

A block of statements preceded by the CASE ELSE statement and followed by the END SELECT statement. The CASE ELSE block is executed under the conditions described above for the CASE ELSE statement.

END SELECT Statement

The END SELECT statement is the last statement of a SELECT block. It must be placed after the SELECT statement, all CASE statements, and the CASE ELSE statement. See "END SELECT Statement" on page 195.

Figure 22 on page 81 shows the logical flow of control in a SELECT block.

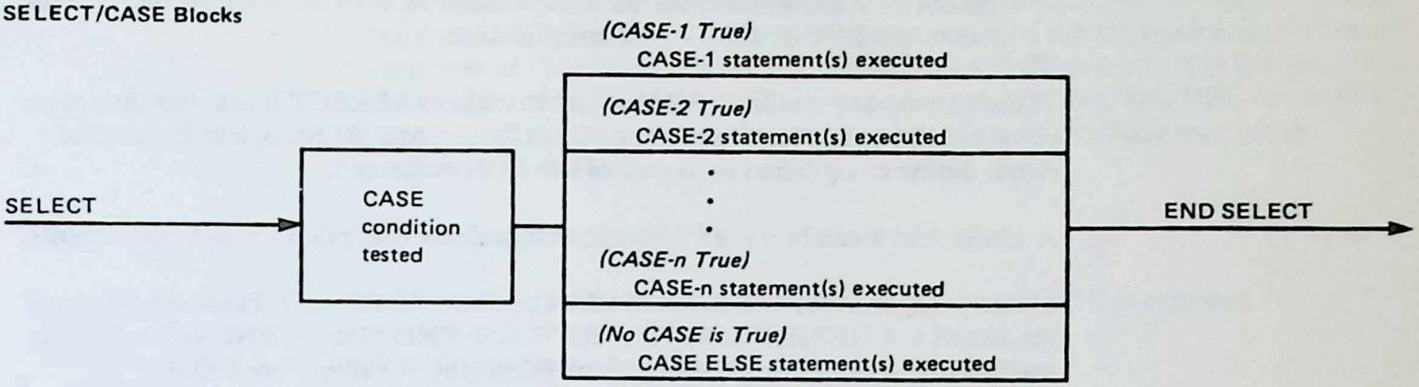


Figure 22. SELECT Block—Flow of Control

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. The expression on line 100 consists of the single variable ABC, whose value is 5. No further evaluation is needed.

2. The range criterion (6 through 10) on line 110 is evaluated and it is determined that 5 is not between 6 and 10, so processing skips to line 150.

3. The inequality criterion (LT 4) on line 150 is evaluated and it is determined that 5 is not less than 4, so processing skips to line 200.

4. The equality criterion (EQ 12) on line 200 is evaluated and it is determined that 5 is not equal to 12, so processing skips to line 250.

5. Line 250 directs execution of its CASE ELSE block in the event that none of the CASE statement criteria are met. Therefore, the CASE ELSE block (lines 260 to the END SELECT statement) is executed.

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, END SUB, or FNEND statement, each having been set originally by a condition (the conditions are GOSUB statements, CALL statements, exceptions, and function references) within that CASE block.

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; 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 assignment statements where the type (decimal, integer, real single, or real double) of the expression to the right of the equal sign does not agree with that of the variable to the left of the equal sign, the result of the expression is converted to the type of the variable on the left. BASIC uses IFIX to convert to integer, SNG to convert to real single, DBL to convert to real double, and DEC to convert to decimal. Rounding occurs when converting; some conversions may cause overflow or underflow.

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 23.

LET A# = 123.56	Assigns decimal or real double constant 123.56 to decimal or real double variable A#.
LET B,C,D = 5.7	Assigns decimal, real single, or real double 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, real single, or real double 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 array ARAY#.

Figure 23. Assignment Statement—Assigning Constant Values

Some examples of assigning one variable to another variable are shown in Figure 24.

Decimal Conversion Rules

As Figure 2 on page 28 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.

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$(2,3)	Defines ABC$ as an array with dimensions 5 by 5. Redimensions ABC$ to 2 by 3, then assigns the values in array XYZ$ to the corresponding elements in array ABC$.

Figure 24. 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. Input/output statements are system dependent. See your IBM BASIC System Services manual for details.

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 on the following page.

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 exceptions to this requirement are the PRINT, PRINT FIELDS, and PRINT File statements, which allow a null list. The null list produces a blank line (or record) of output for PRINT and PRINT File statements and no output for PRINT FIELDS statements.

The input/output list has the following format:

```
[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 (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 in PRINT and PRINT File statements.
- The list may begin and/or 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 (preceding the keyword that identifies the statement) and with no occurrence of MAT in the input/output list.

For example:

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

is equivalent to

```
100 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 items in the list in the order received. If an array appears in the input list, values are assigned to its elements with the rightmost subscripts varying most rapidly. If redimensioning is specified for the array, the redimensioning is done before any values are assigned.

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 exception 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. Values from arrays are written with the rightmost subscripts varying most rapidly. 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 FORM 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.

Input/Output Exception Processing

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

Most of the input/output statements let you specify exception-recovery clauses for that particular statement, either as an EXIT clause or as individual exception clauses, such as CONV, SOFLOW, and IOERR. The EXIT clause, which refers to an EXIT statement, and the other exception clauses are mutually exclusive. Multiple exception clauses are separated by commas and cannot duplicate each other.

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

Data Table Statements

A data table 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 data table statements are:

DATA One or more DATA statements create a data table. (See "DATA Statement" on page 174.)

READ READ statements assign values from the data table to variables and arrays. (See "READ Statement" on page 344.)

RESTORE The RESTORE statement resets the data table pointer to the first value of a specific DATA statement in the data table. (See "RESTORE Statement" on page 358.)

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 full-screen 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. (See "INPUT Statement" on page 243.)

LINE INPUT Provides unformatted data to a program from a terminal for assignment to a character item in an input list. (See "LINE INPUT/LINPUT Statement" on page 263.)

MARGIN Sets the boundaries for writing data via a PRINT statement to a terminal. (See "MARGIN Statement" on page 269.)

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. (See "PRINT Statement" on page 320.)

See also "SET TERM Command" on page 488, and "CALL BASIC Statement" on page 151 for controlling whether line by line input/output is done in SCROLL or LINE mode.

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. (See "INPUT FIELDS Statement" on page 248.)

PRINT FIELDS Displays one or more data values in one or more screen fields. (See "PRINT FIELDS Statement" on page 332.)

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 spaces. 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 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— 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.

Note: Attributes defined for an input field may overlap (and replace) existing attribute characters.

Mixed Mode Operations

Be careful when intermixing full-screen operations with SCROLL mode 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. (See "PRINT Statement" on page 320.)

File Input/Output Statements

File input/output statements are described under "File Input/Output Statements" on page 65, in the "BASIC File Capabilities" on page 61 section.

Program Segmentation Statements

The program segmentation statements segment programs as follows:

- 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.

For more information on internal subroutines, see "Subroutine Control Statements" on page 73, "GOSUB Statement" on page 228, "ON GOTO/GOSUB Statement" on page 303, and "RETURN Statement" on page 361.

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 program units and 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. BASIC supplies interface routines that allow the program to execute operating system commands, to perform Graphical Data Display Manager/Presentation Graphics Feature (GDDM/PGF) operations, to access data in a relational database system (SQL), and to invoke subprograms written in COBOL, FORTRAN, or PL/I and to set the terminal to line or scroll mode.

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. See "DEF Statement" on page 180.

The DEF statement may completely define the function or it may specify the beginning of a function block (multi-statement) 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 multi-statement function. The FNEND statement is the last statement of the block. It serves to mark the end of block and is also the exit point of the function. See "FNEND Statement" on page 202.

A DEF statement, or a DEF/FNEND group of statements may appear anywhere in a program unit, except within another DEF/FNEND group.

Functions are activated by a reference to the function name. Lines in a function definition are not executed unless the function they define is referenced. If execution reaches a DEF statement in some other manner, processing proceeds to the statement 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.

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. Numeric arguments will be converted to the type of the parameter which must be numeric. Character arguments must correspond to character parameters.

Transfer of control into or out of a function other than through function references or by an exception is illegal. If input/output is performed by a function that has been invoked in an input/output data list, then an exception will occur if control is returned to the invoking statement. Unpredictable results may occur 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 particular program unit.

A function definition may not refer, directly or indirectly, to the function being defined (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.

See "Functions" on page 43 for more information.

Single Statement 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.

Example

```
100 DEF MSUB(X,Y,Z) = X * Y - Z
```

Multi-statement Functions

If a function is defined in a DEF block, the statements following the DEF statement are executed in sequential order.

Within multi-statement functions, assignments (LET statements) of values to the function name determine the value to be returned as the value of the function.

If a GOSUB statement within a function branches to another statement within the function and no RETURN statement is executed before the FNEND statement is executed, control returns to the statement which invoked the function. A subsequent RETURN statement will not cause execution to return to the statement after the GOSUB in the function. GOSUB statements which were executed before invoking the function will return normally.

Example

```
100 ! ******** FUNCTION DEFINITION ******
110 !
120   DEF X$
130      X$ = 'FUNCTION VALUE'
140      GOSUB 160
150      STOP                    ! WILL NEVER EXECUTE THIS STATEMENT
160      PRINT 'SUBROUTINE WITHIN FUNCTION ENTERED'
170   FNEND
200 !
210 ! ********* MAIN PROGRAM *************
220 !
500   GOSUB 900
510   PRINT 'RETURN TO MAIN FROM SUBROUTINE 900'
520   GOTO 999
530 !
540 ! ********* SUBROUTINE 900 **********
550 !
900   Y$ = X$                              ! FUNCTION CALLED
910   PRINT 'VALUE RETURNED IS: '; Y$
920   RETURN
930 !
999   END
```

When executed, this program produces the following output:

```
SUBROUTINE WITHIN FUNCTION ENTERED
VALUE RETURNED IS: FUNCTION VALUE
RETURN TO MAIN FROM SUBROUTINE 900
```

Execution proceeds as follows:

1. Lines 100 and 110 are skipped; line 120 then passes control to line 200 (the DEF function is skipped). Lines 200, 210, and 220 are skipped.

2. Line 500: Executes a GOSUB statement saving the return address of 510 and branching to the the first statement of subroutine 900.

3. Line 900: Invokes function X$.

4. Line 130: Assigns the character string 'FUNCTION VALUE' to the function name.

5. Line 140: Executes the GOSUB, saving the return address of 150 and branching to line 160 (within the function).

6. Line 160: Outputs the string 'SUBROUTINE WITHIN FUNCTION ENTERED'. This is the first line of output.

7. Line 170: The end of the function is encountered without having returned from the subroutine call at line 140. The return address of 150 is discarded, and control returns to line 900, which invoked the function.

8. Line 900: Assigns 'FUNCTION VALUE' to Y$.

9. Line 910: Outputs a message including the value of Y$. This is the second line of output.

10. Line 920: Causes control to return to line 510 (not line 160, because it was discarded).

11. Line 510: Outputs a message: 'RETURN TO MAIN FROM SUBROUTINE 900'. This is the third line of output.

12. Line 520: Branches to the end of the program.

Exit from a multi-statement function can only be accomplished by executing the FNEND statement or by an exception. For example, you cannot branch with a GOTO, GOSUB, IF, or ON statement out of the body of a multi-statement function.

Processing a STOP statement in a DEF block ends the entire program (see "Execution Control Statements" on page 82).

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 user-defined function 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.

A main program, if there is one, must precede any subprograms when compiled or run.

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 96).

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. See "SUB Statement" on page 368.

SUBEXIT Ends execution of a subprogram. This statement may occur only in a subprogram. See "SUBEXIT Statement" on page 370.

END SUB Marks the physical end of a subprogram and, if executed, ends execution of the subprogram. See "END SUB Statement" on page 196.

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 also between subprograms (see "CALL Statement" on page 149). A CALL statement is issued in a calling program unit in order to access a called subprogram. Called program units may in turn become calling program units. (See Figure 25 on page 95.)

SUBPRO1 is both a called program unit and a calling program unit. It accepts data from SUBPRO2, processes it, and passes results back to MAINPRO. SUBPRO2 is a called program unit, supplying data to both MAINPRO and SUBPRO1.

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 and the number of 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. All arrays and variables which are not parameters and are not in a COMMON statement are unique to each call.

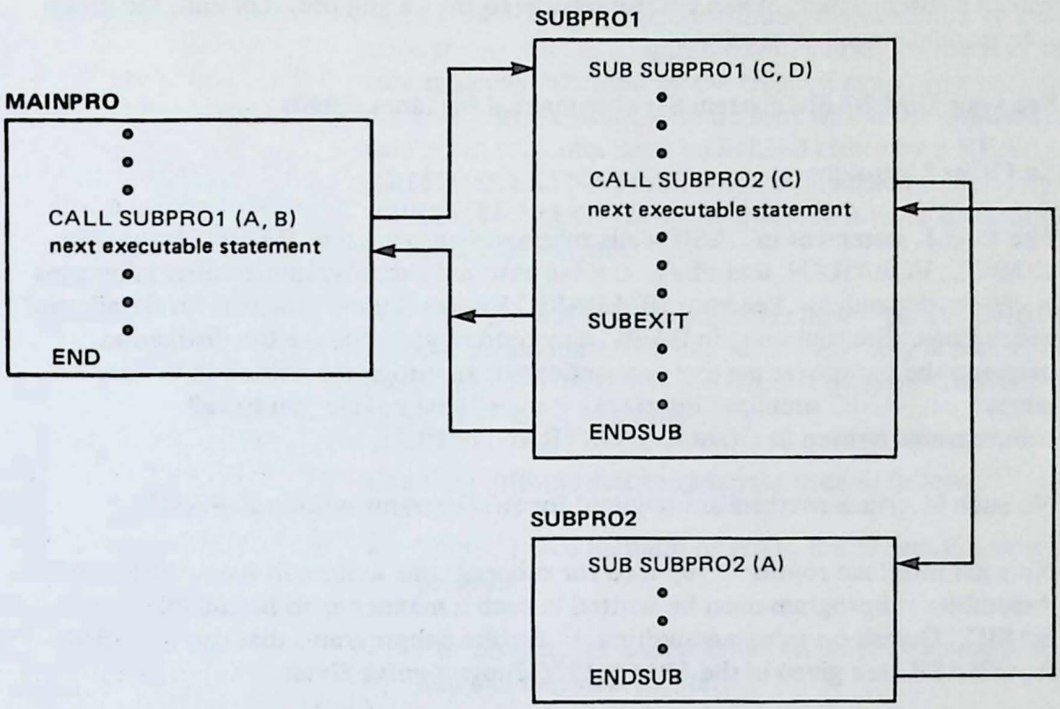


Figure 25. Calling and Called Program Units

Calling BASIC Subprograms

The CALL statement passes control from the calling program unit to the specified subprogram.

The SUB statement is the first line of the subprogram, naming the subprogram and declaring any parameters.

Parameters in a SUB statement may be typed in INTEGER, DECIMAL, REAL, DEFINT, DEFSNG, or DEFDBL statements which come later in the program unit.

When a CALL statement is executed, control transfers from the current program unit to the named subprogram. Execution of the subprogram begins at the line following the SUB statement in the called subprogram and continues until:

- Some other action is dictated by execution of a control statement
- An exception occurs that causes an abnormal termination
- An END SUB, STOP, or SUBEXIT statement is executed

The number and type (and precision of real data) of arguments in a CALL statement must agree with the number and type of parameters in the corresponding SUB statement. There is no type conversion for numeric types (as there is for functions).

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

See your IBM BASIC System Services manual for more details.

Calling Subprograms Written in Other Languages

The CALL statement in BASIC calls subprograms written in BASIC, Assembler, COBOL, FORTRAN, and PL/I. Calling external subprograms in other languages is system dependent. See your IBM BASIC System Services manual for details and restrictions. Programming in BASIC may require you to make the distinction between the parameter passing conventions of subprograms written in these languages. BASIC supplies "interface routines" that enable you to call subprograms written in COBOL, FORTRAN, or PL/I.

No such interface routines are required for subprograms written in BASIC.

No such interface routine is supplied for subprograms written in Assembler. The Assembler subprogram must be written in such a manner as to be callable from BASIC. Details on using and writing Assembler subprograms that can be called from BASIC are given in the *IBM BASIC Programming Guide*.

The interface routines supplied to establish linkage to routines written in COBOL, FORTRAN, and PL/I are: CLINK, FLINK, PLINK, COBOL, FORTRAN, and PLI. See "CALL Statement" on page 149 and "CALL COBOL, FORTRAN or PLI Statement" on page 152 for more information.

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 "CALL SYSTEM Statement" on page 161 for syntax and more information. See your IBM BASIC System Services manual for details and restrictions.

This statement is the analog of the SYSTEM command (see "SYSTEM Command" on page 491).

Chaining Statements

CHAIN Ends execution of the current program (the chaining program) and starts another program (the chained main program). It also specifies which variables are to be passed from the chaining program unit to the chained main program.

USE Specifies which variables the chained main program is expecting to receive from the chaining program unit. A "by name" correspondence must exist: Only those variables that have the same identifiers and attributes in both program units are passed.

The passed attributes for each variable name in the USE statement are matched against the locally defined attributes for the same variable name. An attribute type mismatch results in an unrecoverable error. Note that the passed attributes are not automatically inherited by the chained main program; thus each variable name in the USE statement must be defined within the chained main program. In particular, real data must be defined either by a REAL statement or the option ARITHMETIC NATIVE, with the precision defined by the option {SPREC | LPREC} (or by default); or it must be defined by DEFSNG or DEFDBL statements.

The CHAIN and USE statements allow separate programs to be executed serially, without outside intervention. This capability is useful when segmenting very large programs. See "CHAIN Statement" on page 166 and "USE Statement" on page 373 for more information.

Chaining differs from subprogram calls as follows:

- There is no automatic return to the program unit which executes the chained program.
- The program executed as the result of chaining is in itself a complete program and can be run separately.
- Control passes irrevocably from the program that caused the chained program to be executed. Therefore, the original program is not retained in storage and the space can be used by the chained program.

Figure 26 on page 98 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.

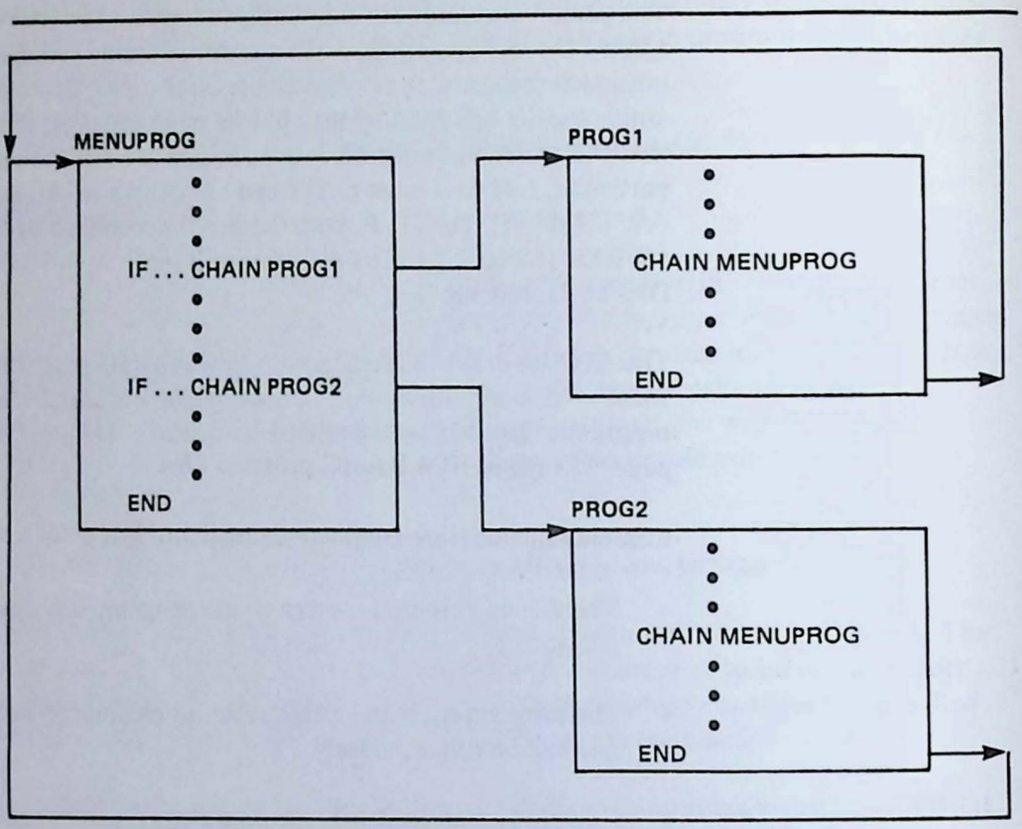


Figure 26. Chaining and Chained Programs

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: Inside the BASIC environment, 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 program unit file with a filename the same as the subprogram name.

If the calling program unit is itself compiled, the subprogram must also be compiled. Thus a CALL within a dynamically loaded compiled program unit file cannot refer back to a subprogram in the workspace.

CHAIN Statement Restrictions: See your IBM BASIC System Services manual for details.

Main programs that are chained to are dynamically loaded. They can be compiled BASIC program files or BASIC source program files.

When a CHAIN statement is encountered inside the BASIC environment, BASIC first attempts to find a compiled program file of the indicated name. If a compiled program file is found, it is loaded and used. If no compiled program file is available, BASIC then attempts to find an BASIC source file, and reload the workspace.

In programs running outside the BASIC environment, CHAIN statements can refer only to compiled IBM BASIC program files.

Program segmentation is system dependent. For details, see your IBM BASIC System Services manual.

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 program units and programs.

A COMMON area created in a main program remains in existence from one subprogram CALL to the next and also from one 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. For more details, see "COMMON Statement" on page 170.

Calling the Graphical Data Display Manager (GDDM)

The Graphical Data Display Manager (GDDM) may be called to perform graphic functions within a BASIC program. The CALL GDDM statement allows you to specify most graphic functions by name (recommended) or by RCP (request control parameter) code.

Use of the Interactive Chart Utility (ICU) is also supported both by name and with a RCP code. It may be called from BASIC using a CALL GDDM statement specifying the CHART function.

You cannot use the following GDDM functions:

```
FSEXIT
FSINIT
FSRNIT
FSTERM
SPINIT
```

These are GDDM initialization and completion functions handled automatically by the BASIC interface routine.

See "CALL GDDM Statement" on page 155 for syntax and details of the CALL GDDM statement. For a more detailed discussion of using GDDM, and for some sample applications, see your *IBM BASIC Programming Guide*.

See the *Graphical Data Display Manager: Base Programming Reference* for more information on GDDM and its functions. The Interactive Chart Utility and the Chart function are discussed in the *Graphical Data Display Manager Presentation Graphics Feature: Programming Reference*, and in the *Graphical Data Display Manager Presentation Graphics Feature: Interactive Chart Utility User's Guide*. For information on how to link to GDDM routines, see your IBM BASIC System Services manual.

Using CALL SQL to Access Relational Data Bases

From your BASIC program, you can access data in IBM's relational data base management systems: DATABASE2 (DB2) in MVS and Structured Query Language/Data System (SQL/DS) in VM. Access to the data is via Structured Query Language (SQL). The data base system executes your SQL statement, performing the requested operation on the data base. If your SQL request requires data to be retrieved from the data base, the appropriate data will be returned to your BASIC program. You can also perform insert, update, and delete operations on a relational data base. See "SQL Statements" on page 101 for the different SQL statements you can use.

In your BASIC program you use the CALL SQL statement to pass a SQL statement to the data base system. The first argument of CALL SQL identifies your SQL statement. The syntax is:

```
CALL SQL (SQL-stmt, status-info(), message-var [,value-list])
```

See "CALL SQL Statement" on page 159 for a detailed description of the syntax.

Note: Since the *status-info* array is used primarily for testing return codes, it is referred to as SQLRC for the sake of convenience in this chapter. The BASE 1 option is assumed in the sections below when referring to elements in the array.

SQL Statements

You can use the following SQL statements in your CALL SQL statement:

- Data Manipulation Statements

  DELETE
  INSERT
  SELECT
  UPDATE

- Data Definition Statements

  ACQUIRE (SQL/DS only)
  ALTER
  COMMENT
  CREATE
  DROP

- Control Statements

  COMMIT
  CONNECT (SQL/DS only)
  EXPLAIN (SQL/DS only)
  LOCK
  ROLLBACK

- Authorization Statements

  GRANT
  REVOKE

In addition, you may use CALL SQL's special SETOPT statement (see "NULL Values" on page 110).

The following SQL statements *cannot be used* in your CALL SQL statement:

- Program Control Statements

  DECLARE TABLE
  INCLUDE
  WHENEVER

- Dynamic Control Statements

  DECLARE STATEMENT
  DESCRIBE
  EXECUTE (and EXECUTE IMMEDIATE)
  PREPARE

- Cursor Related Statements [1]

  CLOSE
  DECLARE CURSOR
  FETCH
  OPEN

In addition to the above SQL statements, the following clauses may not be used in SQL statements:

WHERE CURRENT OF cursor-name

INTO (when used in a SELECT statement)

COMMIT and ROLLBACK

You are responsible for explicitly managing the commit and rollback points within your program using the COMMIT and ROLLBACK SQL statements. These SQL statements are used to make permanent (COMMIT) or "undo" (ROLLBACK) all data base modifications that have occurred since the previous commit or rollback point. It's recommended that your program issue COMMITs and ROLLBACKs as early as practical in order to avoid unnecessary use of data base resources.

BASIC performs an automatic ROLLBACK for any uncommitted work left over when your program ends. Therefore, all data base modifications made since your last commit point will be lost unless you explicitly issue a COMMIT before the program ends.

COMMITs are usually unnecessary for SQL statements that do not modify the data base (for example, SELECT). See "Deactivating an Active SQL Statement" on page 110.

CALL SQL Statements With and Without Value-lists

You need to specify a *value-list* when your CALL SQL statement includes a SELECT statement. A value-list is also required if your CALL SQL includes INSERT, DELETE, or UPDATE statements which use index values. (Index values in SQL statements are discussed on page 103.)

For SELECT statements, the elements of the value-list correspond to the column names in your SQL statement. For example, in the CALL SQL statement:

```
130 CALL SQL (STMT$,SQLRC(),SQLMSG$,EMPL$,DEP$,SAL)
```

the value-list consists of EMPL$, DEP$, and SAL which are variables in your BASIC program. After the CALL SQL statement is executed, the required data from the relational data base will be found in EMPL$, DEP$, and SAL.

---

[1] Note that these SQL statements are issued *internally* by BASIC. For example: a SELECT statement is assigned a predeclared statement and cursor name; before any rows are retrieved, the statement is described (via the DESCRIBE statement) and the cursor is opened (via OPEN); each row is retrieved with a FETCH statement; at table end the CLOSE statement closes the cursor.

The SQL statement you wish to pass to the data base system is stored in the first argument of CALL SQL; for example:

```
100 STMT$="SELECT NAME,DEPT,SALARY FROM EMPLOYEE WHERE SALARY > 20000"
```

Since SQL users commonly write each clause of a SQL statement on a new line, you could conceptually think of the above SQL statement as being:

```
SELECT NAME, DEPT, SALARY
FROM EMPLOYEE
WHERE SALARY > 20000
```

where NAME, DEPT, and SALARY are column names of the EMPLOYEE table. You could store the SQL statement into STMT$ as follows:

```
100 STMT$='SELECT NAME,DEPT,SALARY '
110 STMT$=STMT$ & 'FROM EMPLOYEE '
120 STMT$=STMT$ & 'WHERE SALARY > 20000'
```

After the CALL SQL statement is executed, appropriate data from the NAME, DEPT, and SALARY columns of the EMPLOYEE table will be stored in EMPL$, DEP$, and SAL respectively.

How value-lists are used for statements which use index values is discussed in "Index Values in SQL Statements."

You would use the CALL SQL statement without a value-list when you want to execute a SQL statement that does not produce a result table or does not use index values. This can apply to the following SQL statements:

ACQUIRE (for SQL/DS only)
ALTER
COMMENT
COMMIT
CONNECT (without index values, for SQL/DS only)
CREATE
DELETE (without index values)
DROP
EXPLAIN (for SQL/DS only)
GRANT
INSERT (without index values)
LOCK
REVOKE
ROLLBACK
UPDATE (without index values)

Index Values in SQL Statements

The SQL language allows you to execute a particular SQL statement many times while providing different values for clauses such as WHERE and VALUES. These values, instead of being "hard-coded" in the SQL statement, are supplied in program variables, termed "host variables."

BASIC uses a similar concept. A host variable is specified as an index to the list of program variables, in the "value-list." This index is preceded by a colon (similar to the way host variables are specified for application programs in DB2 or SQL/DS).

The index is an integer which specifies the position of the value within the value-list (the first one is specified as :1). Before BASIC passes data or retrieval requests to the data base system, the index is effectively replaced by the appropriate value from the value-list. For example, suppose the following SQL statement:

```
INSERT INTO EMPLOYEE
  (NAME,DEPT,SALARY,BRTHDATE)
  VALUES (:1,:2,:3,:4)
```

is stored in a variable STMT$ as follows:

```
100 STMT$='INSERT INTO EMPLOYEE '
110 STMT$= STMT$ & '(NAME,DEPT,SALARY,BRTHDATE) '
120 STMT$= STMT$ & 'VALUES (:1,:2,:3,4)'
```

(where NAME, DEPT, SALARY, and BRTHDATE are column names in the data base).

Now suppose your BASIC program stores the values JOHN SMITH, J38, 42500, and 09/15/45 in NAME$, DEPT$, SALARY, and BIRTHDATE$. Your CALL SQL statement would look like this:

```
CALL SQL (STMT$,SQLRC(),SQLMSG$,NAME$,DEPT$,SALARY,BIRTHDATE$)
```

When the SQL statement is received by the data base system, the index values in STMT$ will have been replaced with the corresponding values from the value-list, NAME$, DEPT$, SALARY, and BIRTHDATE$. Thus the SQL statement which the data base system receives will be equivalent to:

```
INSERT INTO EMPLOYEE
  (NAME,DEPT,SALARY,BRTHDATE)
  VALUES ('JOHN SMITH','J38',42500,'09/15/45')
```

The items in the value-list are, in effect, an argument list for the index values in the SQL statement. Notice that the first index value (:1) receives the first argument in the value-list (NAME$), the second index value (:2) gets the second item of the value-list (DEPT$) and so on. Index values can be used in the following clauses:

HAVING
SET
VALUES
WHERE
CONNECT (SQL/DS only)

See Example 2: Inserting Rows into a Table on page 124 for an illustration of using index values.

SQL Operations

This section describes how the CALL SQL statement is used for the different SQL statements you can pass to the data base system. The following types of SQL statements are considered (the statement types are discussed in order of complexity—from simple to complex):

- Nondata Manipulation statements

- DELETE
- UPDATE
- INSERT
- SELECT

Nondata Manipulation SQL Statements

Nondata manipulation SQL statements include all SQL statements you can use in BASIC with the exception of SELECT, INSERT, DELETE, and UPDATE. These SQL statements are shown on page 101, grouped under data definition statements, control statements, and authorization statements. Some examples are:

```
100 CALL SQL ('COMMIT WORK', SQLRC(), SQLMSG$)

200 CALL SQL ('DROP VIEW SPECIAL1', SQLRC(), SQLMSG$)

300 STMT$="GRANT SELECT ON STOCKS TO JONES"
310 CALL SQL (STMT$, SQLRC(), SQLMSG$)

400 STMT$="REVOKE UPDATE ON TABLE1 FROM JEFFERS"
410 CALL SQL (STMT$, SQLRC(), SQLMSG$)
```

The CONNECT statement may be used with or without index values.

Examples

```
100 CALL SQL ("CONNECT PAULA IDENTIFIED BY X2YM85", SQLRC(), SQLMSG$)

200 PW$= "X2YM85"
210 CALL SQL ("CONNECT PAULA IDENTIFIED BY :1", SQLRC(), SQLMSG$, PW$)

300 N$= "PAULA": PW$= "X2YM85"
310 CALL SQL ("CONNECT :1 IDENTIFIED BY :2", SQLRC(), SQLMSG$, N$, PW$)
```

DELETE Statements

The CALL SQL statement which passes a DELETE statement to the data base system may or may not have a value-list. It depends on whether or not index values are used in the WHERE clause of the SQL statement. The two CALL SQL statements in the example below show how a DELETE statement can be passed to the data base system; the first SQL statement uses "hard-coded values" and doesn't need a value-list, the second uses index values and a value-list:

```
600 DIM STMT$*256
610 STMT$="DELETE FROM EMPLOYEE WHERE "
620 STMT$=STMT$ & "NAME='BROWN' AND DEPT='SALES'"
630 CALL SQL (STMT$, SQLRC(), SQLMSG$)
.
.
.
700 STMT$="DELETE FROM EMPLOYEE WHERE "
710 STMT$=STMT$ & "NAME=:1 AND DEPT=:2 "
720 LET N$="BROWN"
730 LET D$="SALES"
740 CALL SQL (STMT$, SQLRC(), SQLMSG$, N$, D$)
```

UPDATE Statements

The CALL SQL statement which passes an UPDATE statement to the data base system may have a value-list, depending on whether or not index values are used in the SET and/or WHERE clauses. The following examples illustrate using index values and value-lists for UPDATE:

```
100 STMT$="UPDATE EMPLOYEE SET DEPTNO=:1 WHERE DEPT='MFG'"
110 DNO$="J38"
120 CALL SQL (STMT$, SQLRC(), SQLMSG$, DNO$)
.
.
.
200 STMT$="UPDATE EMPLOYEE SET SAL=SAL+50 WHERE DEPT=:1"
210 D$="SALES"
220 CALL SQL(STMT$,SQLRC(),SQLMSG$,D$)
.
.
.
300 STMT$="UPDATE EMPLOYEE SET JOBCODE=:1 WHERE NAME=:2"
310 JC=54
320 N$="JONES"
330 CALL SQL(STMT$,SQLRC(),SQLMSG$,JC,N$)
```

Single-Row and Multiple-Row Access

The CALL SQL statement enables you to insert or retrieve data a single row at a time or multiple rows at a time. Multiple-row access applies to INSERT and SELECT statements only.

The INSERT statement inserts rows into a table and the SELECT statement retrieves rows from a table. Each execution of CALL SQL with an INSERT statement will insert either a single row or multiple rows into a table. Similarly for SELECT, each execution of CALL SQL will retrieve either a single row or multiple rows. The number of rows inserted or retrieved in a multiple-row operation is called a *segment*. Segmenting is only allowed for INSERT and SELECT statements and requires the use of arrays instead of scalars in the value-list. How you specify whether you want a single row or multiple rows for each execution of CALL SQL differs, depending on whether your SQL statement is INSERT or SELECT and is described with each statement below.

INSERT Statements

In SQL the INSERT statement can be used in two different forms:

- With a subquery
- With a VALUES clause

When the CALL SQL statement passes an INSERT with a subquery, you can only have a value-list for the WHERE clause. For example:

```
100 DMFG$='MFG'
110 DSALES$='SALES'
120 STMT$='INSERT INTO NEWTABL SELECT NAME,DEPT,SALARY,BRTHDATE '
130 STMT$= STMT$ & 'FROM EMPLOYEE WHERE DEPT=:1 OR DEPT=:2'
140 CALL SQL (STMT$,SQLRC(),SQLMSG$,DMFG$,DSALES$)
```

Note that even though the execution of this example may result in several rows being inserted into the table, it is not a "segmented" operation as described in "Single-Row and Multiple-Row Access." Segmenting is not allowed for INSERT with a subquery. The variables in the value-list correspond to the WHERE clause and must be scalars.

When the INSERT statement has a VALUES clause, the CALL SQL statement may use a value-list to provide values to be inserted into a table. The value-list corresponds to the index values in the VALUES clause. For example:

```
100 N$='SMITH'
110 D$='ACCOUNTING'
120 S=44000
130 B$='10/10/42'
140 STMT$='INSERT INTO NEWTABL (NAME,DEPT,SALARY,BRTHDATE) '
150 STMT$=STMT$ & 'VALUES (:1,:2,:3,:4)'
160 CALL SQL (STMT$,SQLRC(),SQLMSG$,N$,D$,S,B$)
```

The above example will insert a single row into the table. Note that the elements of the value-list, N$, D$, S, B$ are scalars. However, the use of INSERT with a VALUES clause can be segmented; which means that multiple rows can be inserted each time the CALL SQL statement is executed. To insert multiple rows, the value-list variables must be defined as *one-dimensional arrays* rather than scalars. The size of the segment depends on the value stored in SQLRC(3)— the third element of the status info array:

- If SQLRC(3) is zero, then the current dimension of the value-list arrays is used.
- If SQLRC(3) is positive and less than or equal to the current dimension of the value-list arrays, then the value stored in SQLRC(3) is used.
- If SQLRC(3) is negative or greater than the current dimension of the arrays, an error code is returned in SQLRC(2).

An INSERT operation is considered segmented when its value-list consists of arrays, even if the dimension of those arrays is 1.

The current dimensions of all arrays comprising the value-list must be equal. The following example illustrates how 20 rows at a time can be inserted:

```
 90 OPTION BASE 1
100 DIM N$(20), D$(20), S(20), B$(20)
110 SQLRC(3)=0
120 STMT$='INSERT INTO NEWTABL (NAME,DEPT,SALARY,BRTHDATE) '
130 STMT$=STMT$ & 'VALUES (:1,:2,:3,:4)'
.
.
.
200 CALL SQL (STMT$, SQLRC(), SQLMSG$, N$(), D$(), S(), B$())
```

In the above example the segment size is 20 rows because the current dimension of each value-list array is 20 elements and because SQLRC(3) is zero. A segment of less than 20 rows can be obtained by setting SQLRC(3) to any positive value less than 20. See "The status-info Array" on page 114 for more information about SQLRC(3).

The example assumes that appropriate data will be stored in the value-list arrays before the CALL SQL statement is executed.

SELECT Statements

In SQL, SELECT statements retrieve data from a table. When a CALL SQL statement contains a SELECT statement, a value-list is always required for receiving data. The elements of the value-list correspond to the column names in the SELECT statement.

Retrieval can be single row at a time or multiple rows at a time. For single row retrieval, the value-list consists of variables. For multiple row (segmented) retrieval, the value-list contains one-dimensional arrays all having the same dimension. The number of rows in a segment for segmented retrieval depends on the value stored in SQLRC(3).

- If SQLRC(3) is zero, then the current dimension of the value-list arrays is used.
- If SQLRC(3) is positive and less than or equal to the current dimension of the value-list arrays, then the value stored in SQLRC(3) is used.
- If SQLRC(3) is positive and greater than the current dimension of the arrays, an error code is returned in SQLRC(2).
- If SQLRC(3) is negative, no rows are returned and the SQL statement is "deactivated." See "Deactivating an Active SQL Statement" on page 110.

A SELECT operation is considered segmented when its value-list contains arrays, even if the dimension of those arrays is 1.

Note: The current dimension of all arrays in the value-list must be equal.

For example, for a CALL SQL statement to retrieve a segment of 25 rows at a time, the DIM statement would be:

```
110 DIM NAME$(25)*30, DEPT$(25)*3, SALARY(25), BIRTHDATE$(25)
```

The Example 4 on page 126 illustrates how you might retrieve a 5-row segment.

The SELECT statement is the only statement in which you can use the value-list both to pass information to the data base system *and* to receive data from the data base.

When discussing the value-list in "CALL SQL Statements With and Without Value-lists" on page 102, the following SQL query was considered:

```
SELECT NAME, DEPT, SALARY
FROM EMPLOYEE
WHERE SALARY > 20000
```

This example could be modified so that instead of "hard coding" the value 20000 in the statement, you could pass any salary value via a variable. The SQL statement becomes:

```
SELECT NAME, DEPT, SALARY
FROM EMPLOYEE
WHERE SALARY > :4
```

The CALL SQL statement becomes:

```
300 CALL SQL (STMT$,SQLRC(),SQLMSG$,EMPL$,DEP$,SAL,AMOUNT)
```

Notice that a fourth element (AMOUNT) has been added to the value-list. AMOUNT is a variable in the BASIC program, and before the data base system is called, :4 will effectively be replaced by the value of AMOUNT. Thus if AMOUNT is set to 30000 before execution, the SQL statement passed to the data base system will be equivalent to:

```
SELECT NAME, DEPT, SALARY
FROM EMPLOYEE
WHERE SALARY > 30000
```

The first three elements of the value-list in the above example may be either scalars or arrays as explained earlier. The fourth element, which specifies a condition that must be true of *all* rows returned, is used only once when the SQL statement is first passed to the data base system. It must be scalar.

Value-list elements which correspond to index values used in subclauses such as WHERE or HAVING must follow value-list elements which are used for receiving data from the data base system.

"Active" SQL Statements

When a CALL SQL statement with a SELECT operation is executed, a single row or a segment of rows is retrieved from a table. If you were to execute that very same statement again, the next row or segment in the table would be retrieved. This sequence would continue until an end-of-table condition is detected. While rows are being retrieved, the SQL statement is considered *active*. After reaching the end of table, it is no longer active. If you were to execute the same statement after reaching end of table, it would be considered a new query. BASIC allows up to 20 SQL statements to be active at a time.

The concept of an active statement implies that two *identical* SQL statement strings sent to the data base system represent the *same* statement. Therefore, if you really want two identical SELECT operations to be active at the same time, you must somehow make them look different (perhaps by adding an extra space or two in the body of one of the statements; don't use leading or trailing spaces because they're ignored).

Remember that index values used in such clauses as WHERE and HAVING apply to *all* rows returned by an active statement. Changing such values after the first and before the last row is retrieved will have no effect. In fact, for subsequent executions of the same SELECT before the last row is retrieved, these value-list elements can be omitted; if present, they are checked for errors but otherwise ignored. However, once the last row has been retrieved, if the same SELECT is executed again, these elements are reexamined and their now current values take effect.

Deactivating an Active SQL Statement

Besides reaching end of table, there are two ways of explicitly deactivating an active SQL request:

- Issuing a COMMIT (or ROLLBACK) statement. This closes *all* cursors and deactivates *all* active statements.
- Assigning a negative value to the third element of the status info array (SQLRC(3)) and reexecuting the CALL SQL statement. This effectively tells BASIC that you don't want any more rows from the table. (The value-list is not required when SQLRC(3) is negative.)

Again, if the same SELECT were to be executed after either of the above actions, the second SELECT would be considered a new query.

NULL Values

In the SQL language a *null value* indicates the absence of a value in a table. It is not the same as zero, a blank space, or a string of length zero; rather, it represents a value that is unknown, or a value that has not been assigned yet. The SQL language uses a special keyword (NULL) to refer to null values. However, NULL cannot be used in an index variable to represent NULL data. Therefore, a previously agreed to value is used to represent null values. The default value is:

- For character strings: an empty string
- For numeric values: -2147483648

The above numeric value applies to all numeric data types (decimal, integer, real single, or real double).

The selected numeric null value (-2147483648) is a default and you can explicitly redefine it by executing a special form of the CALL SQL statement as shown below:

Format

```
CALL SQL (null-def, status-info(), message-var [,value-list]
```

where:

null-def
is a character expression which evaluates to:

```
SETOPT NULL n
```

where:

n
is an integer from +2147483647 to -2147483648 or is an index variable.

value-list

if an index variable is used, it must refer to a value-list element of integer type which contains the represented NULL value.

Examples of CALL SQL to redefine the default numeric representation of NULL are:

```
100 CALL SQL ("SETOPT NULL -1000000", SQLRC(), SQLMSG$)
```

or

```
200 LET SETNULL$="setopt null 0"
210 CALL SQL (SETNULL$, SQLRC(), SQLMSG$)
```

or

```
300 INTEGER NULLVAL
310 LET NULLVAL=-1
320 CALL SQL ("SETOPT NULL :1",SQLRC(),SQLMSG$,NULLVAL)
```

This form of CALL SQL is unique to BASIC and is not passed to the data base system. You can only use it to redefine numeric null values. The null value for character data is always an empty string.

It is possible that a represented null value (either the default or one chosen by you) may be equal to and therefore conflict with an actual value being retrieved from the data base. In such an event, the fourth and fifth elements of the status-info array are set to nonzero values, identifying where the first conflict occurred. See "The status-info Array" on page 114 for more information.

Index values referring to represented nulls (or to zero-length strings) can be used only in the VALUES clause of INSERT or the SET clause of UPDATE. Use of such index values in other contexts is an error.

If you wish to INSERT a zero-length string or the numeric value of the current represented null, you must "hard-code" it in your SQL statement. The same is true when referring to represented null values in the WHERE clause.

Thus, in the sequence

```
100 A$ = '' : B = 0          ! A$ is zero length character string
110 CALL SQL ("SETOPT NULL 0", SQLRC(), SQLMSG$)
120 CALL SQL ("INSERT INTO T (CHAR_COL, INT_COL) VALUES (:1, :2)",  &
  &           SQLRC(), SQLMSG$, A$, B)
130 CALL SQL ("INSERT INTO T (CHAR_COL, INT_COL) VALUES ('', 0)",  &
  &           SQLRC(), SQLMSG$)
```

Statement 120 inserts null into both columns; statement 130 inserts non-null values into both statements.

Also,

```
140 CALL SQL ("UPDATE T SET CHAR_COL = :1 WHERE INT_COL = 0",       &
  &           SQLRC(), SQLMSG$, A$)
```

will change the zero-length string inserted by statement 130 to a SQL null.

But,

```
150 CALL SQL ("DELETE FROM T WHERE INT_COL = :1",                    &
  &           SQLRC(), SQLMSG$, B)
```

will result in an error (represented nulls cannot be used in WHERE clause value-list elements).

On the other hand,

```
160 CALL SQL ("DELETE FROM T WHERE INT_COL IS NULL",                 &
  &           SQLRC(), SQLMSG$)
```

will successfully delete the row inserted by statement 120.

A Caution When Using Real Data

Note: BASIC will consider a value to be null only if it *exactly* matches the null representation, which must be an integer. Values of other types will not be considered null if they contain a decimal fraction.

The distinction is important in the case of real data. Suppose your SETOPT statement had been "SETOPT NULL 1." Then a real single or real double variable containing, for example, 1.25E0—which *converts* to the integer 1— would not be considered null because its internal representation, rather than hexadecimal '4010' followed by 4 (or 12) zeroes, would be '4014' followed by 4 (or 12) zeroes.

The above "exact match" rule leads to a special restriction for real single variables: when the selected null representation (including the default value of -2147483648) has an absolute value greater than 16777215, any real single variable sent to or retrieved from the data base whose absolute value is greater than 16777215 will cause execution of the SQL statement to be halted with an error indication. In order to retry the operation you will have to execute SETOPT again specifying a smaller integer. Consequently, in a BASIC program employing a wide range of real single data, you might be well advised to convert the data to real double (or decimal) form before executing any SQL Data Manipulation Statements.

Data Types and Conversions

BASIC ensures that arguments passed to the data base system are of a compatible format. Some data formats are unique to BASIC while some others are unique to the data base system. When necessary, BASIC converts to the closest common format.

For example, suppose the data type for the variable SALARY in your BASIC program is DECIMAL and suppose the data type for the SALARY column in the data base is DECIMAL(m,n). These two data types are different. BASIC DECIMAL is a floating point data type whereas DECIMAL(m,n) is a fixed point decimal. When data passes from the data base system to BASIC, the SALARY value in DECIMAL(m,n) will be converted to BASIC DECIMAL by BASIC.

Similarly, when a DECIMAL SALARY value is being passed from BASIC to the data base system, BASIC will convert the value to DECIMAL(m,n). It's possible that this conversion may result in an overflow condition and/or loss of precision.

When the CALL SQL statement is executed, data may pass in both directions—from BASIC to the data base system and vice versa. Conditions such as overflow, truncation, or rounding will depend on the direction in which the data is being transferred. The two figures, Figure 27 and Figure 28 on page 114 illustrate data conversion and possible errors or loss of accuracy. (For more information about BASIC data types and internal representation, see "Data: Constants, Variables, and Arrays" on page 21.)

Data Base System Data Types	BASIC Data Types				
	DECIMAL	INTEGER	REAL SINGLE	REAL DOUBLE	CHAR
DECIMAL(m,n)	X	I,1,2	F,D,2	F	error
SMALLINT	I,X	I	F,D	F	error
INTEGER	X	*	F,D	F	error
FLOAT	X	I,1,2	D,2	*	error
CHAR(n)	error	error	error	error	3
VARCHAR(n)	error	error	error	error	3
GRAPHIC and VARGRAPHIC	error	error	error	error	error

```
Legend:  * - Identical data types
       Conversions by the data base system:
         F - to FLOAT
         I - to INTEGER
       Conversions by BASIC:
         D - REAL DOUBLE to REAL SINGLE
         X - to BASIC DECIMAL
       Effects and conditions:
         1 - Rounded to integer
         2 - Possible overflow condition
         3 - Truncation possible
Notes: Since exception conditions such as overflow and truncation
       take  place within the data base system, their occurrence
       cannot be tested in the BASIC program with the ON
       Condition statement.

       Whatever applies to VARCHAR and VARGRAPHIC applies to LONG
       VARCHAR and LONG VARGRAPHIC also.
```

Figure 27. Data Transfer from the Data Base System to BASIC

BASIC Data Type	Data Base System Data Types						
	SMALLINT	INTEGER	DEC(m,n)	FLOAT	CHAR(n)	VARCHAR	GRAPHIC, VARGRAPHIC
DECIMAL	X,C,1,2	X,C,1,2	X, 2,5	X,C,2	error	error	error
INTEGER	C, 2	*	C	C	error	error	error
REAL SINGLE	D,C,1,2	D,C,1,2	D,C,2,5	D	error	error	error
REAL DOUBLE	C,1,2	C,1,2	C,2,5	*	error	error	error
CHAR	error	error	error	error	3,4	*,3	error

```
Legend:  * - Identical data types
       Conversion by BASIC:
         D - REAL SINGLE to REAL DOUBLE
         X - BASIC DECIMAL  to data base DECIMAL (DECIMAL(m,n))
       Conversions by the data base system:
         C - conversion to final desired form
       Effects and conditions:
         1 - Rounded to integer
         2 - Possible overflow condition
         3 - Truncation possible
         4 - Pad with spaces if necessary
         5 - Possible loss of precision
Notes: Since exception conditions such as overflow and truncation take
       place within the data base system, or are intercepted by BASIC's
       SQL interface before they occur, their occurrence cannot be
       tested in the BASIC program with the ON Condition statement.

       Whatever applies to VARCHAR and VARGRAPHIC applies to LONG
       VARCHAR and LONG VARGRAPHIC also.
```

Figure 28. Data Transfer from BASIC to the Data Base System

The status-info Array

SQL information and return codes are passed to your BASIC program through the *status-info* argument of the CALL SQL statement. This 20-element integer array is shown below (using the BASE 1 option). The first seven elements contain information specific to BASIC. The next 13 either contain information returned from the data base system, or are reserved for that purpose.

Note: If BASIC detects an error in the first three CALL SQL arguments (for example, if the *status-info* argument is not an integer array of at least 20 elements), it generates a subprogram parameter "mismatch" exception condition, which can be tested with the ON Condition statement.

Element	Description
Element 1	CALL SQL status code. This is a general status code for the CALL SQL statement. The values which can be returned and what they mean are: 0 - Successful completion without warnings or errors 4 - BASIC or SQL warnings. There are three types: • BASIC warning in element 2 • Conditions flagged in elements 10 through 17 • Positive value in element 8 8 - BASIC error detected (see element 2) 12 - Data base error detected (see element 8) 16 - System Error (exception condition)—can be tested with the ON CONDITION statement (*Note*: There is a direct relationship between this code and other error and warning conditions detected by BASIC or the data base system. Refer to "Return Code Relationships" on page 118 to see how this code relates to other codes.
Element 2	CALL SQL error code. A nonzero value indicates an error condition detected by BASIC or an end-of-table condition for a SELECT. (Note: A value of 100 in this element along with a value of 8 in element 1 indicates an end-of-table condition for SELECT statements.) See Appendix B, "CALL SQL Error Codes" on page 503 for the list of SQL interface error codes. For system errors, the value of the CALL SQL error code is assigned to the CODE function. See "CODE Code" on page 385.
Element 3	Table row count. This field has meaning only for SELECT and INSERT...VALUES operations. It applies to both the segmented and non-segmented forms. It is ignored for all other operations, including the subquery form of insert. Before execution of CALL SQL, a **zero:** – is ignored for non-segmented operations. – means that the number of rows to be retrieved or inserted is determined by the dimension of the arrays in the value-list of a segmented operation.

positive value:

- is ignored for non-segmented operations.
- defines the segment size, that is, the number of rows to be retrieved or inserted for each CALL SQL. If this value is larger than the current dimension of the value-list arrays, BASIC signals an error.

negative value:

- is an error with INSERT.
- indicates an explicit end to the SELECT operation (it tells BASIC you want the current SELECT operation to be deactivated and to return no more rows).

After execution of CALL SQL, this field contains the actual number of rows retrieved or inserted. This number may be

zero:

- if an error was detected before the first row was processed or during processing of the first row.
- if no row was retrieved for a SELECT, that is, if the value before SELECT execution was negative, or if end-of-table occurred before any row was retrieved.

  Note that, with SELECT, if *n* rows are requested and **exactly** *n* rows are available, SQLRC(3) will contain *n*. It is the **next** execution of the SELECT that will result in a SQLRC(3) of zero, because no more rows are available.

one:

- for a successful non-segmented SELECT or INSERT...VALUES operation.

equal to the number of rows requested

- if all requested rows were successfully retrieved/inserted.

less than the number of rows requested

- if an error was detected or end-of-data on SELECT occurred before all requested rows could be retrieved or inserted.

Element 4 NULL conflict row indicator.

For segmented SELECT operations, a nonzero value indicates that a numeric value which represents a NULL is in conflict with an actual row number (relative to 1) where the conflict occurred.

If there is a NULL conflict for a non-segmented SELECT operation, the value in this field will be 1.

(Note that return of a zero-length string is *not* marked as a null conflict.)

Null conflict is not considered an error. Execution continues.

Element 5 NULL conflict column indicator.

For segmented and non-segmented SELECT operations, a nonzero value indicates that a value that represents a NULL is in conflict with an actual value returned from the data base. This field contains the position in the value-list where the first conflict occurred. (For example, if the field contains a 2, it means the second value-list argument is in conflict.)

Element 6 Reserved (should be zero)

Element 7 Reserved (should be zero)

Elements 8 through 20 represent information obtained from a control block of the data base system called the SQL Communication Area (SQLCA). The SQLCA itself is not accessible to your BASIC program. If you need more information about the SQLCA, see *IBM DATABASE 2 Application Programming Guide* or *SQL/DS Application Programming*. Also, more information about SQL return codes from the data base system can be found in *IBM DATABASE 2 Messages and Codes* or *SQL/DS Messages and Codes*.

Element 8 SQL return code from the data base system (SQLCODE).

Element 9 Table modification indicator (SQLERRD3).

This indicator tells you how many rows were inserted, modified, or deleted after an INSERT, UPDATE, or DELETE operation in the data base. Note that, for a segmented INSERT this indicator will always be 1. (BASIC may do multiple data base calls, but the data base system inserts only 1 row per call.)

Element 10 Global warning indicator (SQLWARN0).

If this indicator is 1, it means that at least one of the elements that follow (element 11 through element 20) is 1. Otherwise, it is zero.

For segmented operations, Element 10 and Elements 11 through 14 reflect warnings applicable to any or all rows of the segment.

Element 11 Truncation warning indicator (SQLWARN1).

This indicator is set to 1 if at least one column's value was truncated when returned from the data base system. Otherwise, it is zero.

Element 12 Null warning indicator (SQLWARN2).

If set to 1, it means that at least one NULL value was eliminated in the computation of an intrinsic function (NULLs were ignored). Otherwise, it is zero.

Element 13 Mismatch indicator (SQLWARN3).

Note: SQLWARN3 is not set in the status info array. BASIC detects the mismatch between number of columns and number of value-list elements and returns a BASIC error in SQLRC(2) without sending the statement to the data base system.

Element 14 UPDATE or DELETE indicator (SQLWARN4).

If set to 1, it indicates that an UPDATE or DELETE statement did not include a WHERE clause. Otherwise, it is zero.

Element 15 SQL/DS warning indicator (SQLWARN5).

If set to 1, it indicates that a SQL statement submitted to DB2 is valid only in SQL/DS.

Element 16 SQL/DS logical unit of work termination indicator (SQLWARN6).

If set to 1, it indicates that SQL/DS detected a possible deadlock situation as a result of the last SQL statement executed. This indicator is not set after a ROLLBACK WORK statement, only when an implicit rollback is performed by the system.

If set to 2, it indicates that SQL/DS has issued a "severe" SQLCODE (Element 8 - SQLCODE). This severe code indicates that SQL/DS is in an unusable state. Any attempts to further access SQL/DS will cause the application to be abnormally terminated.

Element 17 SQLWARN7

Elements 18-20 Reserved

Return Code Relationships

As indicated earlier, there is a direct relationship between the CALL SQL status code, and error and warning conditions found by BASIC or the data base system. The CALL SQL status code (SQLRC(1)) is an indicator of what has happened as a result of the last SQL request, and it directs you where to look to determine what has occurred as a result of the last CALL SQL request.

This relationship is illustrated in Figure 29 on page 119.

Condition	CALL SQL Status Code SQLRC(1)	Call SQL Error Code SQLRC(2)	SQL Return Code SQLRC(8)
No errors or warnings	0	0	0
Warnings	4	>=0	>=0
SELECT end-of-table	8	100	100
Error detected by BASIC	8	nonzero	0
Error detected by data base	12	0	< 0
System error	16	nonzero	any

Figure 29. Return code relationships

The first row shows that when SQLRC(1) is zero, both the SQL return code and the CALL SQL error code are zero.

The second row shows that when SQLRC(1) = 4, either SQLRC(2) or SQLRC(8) may contain a warning code. Warning codes indicate conditions that are normally ignored but that may have relevance for certain programs. Such conditions do not halt execution of a statement.

CALL SQL warnings are indicated by a nonzero value in SQLRC(2). Data base warnings are indicated either by a positive value in SQLRC(8) or a 1 in SQLRC(10) and one or more of the other warning indicators.

An end-of-table warning condition cannot usually be ignored and is therefore given special treatment by BASIC.

The third row shows the special case when an end of table condition occurs during a SELECT operation. For this situation, a CALL SQL status code of 8 indicates that you must look at the CALL SQL error code, SQLRC(2). This element will be nonzero as expected. During SQL processing, the data base system may return a 100 return code which indicates "row not found"; for SELECT operations, this means an end of table condition has occurred. The 100 code is then echoed in the CALL SQL error code element, SQLRC(2), to reflect the same end of table for SELECT operations. When a data base return code of 100 occurs with UPDATE or DELETE operations, it indicates a "row not found" condition and is considered a warning from the data base system. Hence SQLRC(1) contains 4, SQLRC(2) is 0, and SQLRC(8) is 100.

The fourth, fifth, and sixth rows reflect error conditions in the data base system or BASIC. In general, nonzero values for SQL return code and CALL SQL error return codes are mutually exclusive, with data base error conditions having precedence over BASIC error conditions. System errors (SQLRC(1)=16) cause exception conditions and can be tested with the ON CONDITION statement.

SQL Messages

Messages from BASIC and the data base system are returned to your BASIC program as character strings. The message area is a character variable identified by the third argument of the CALL SQL statement. The maximum length of a message is 256 characters. The returned message will be truncated from the right if the length of your character variable is smaller than the message being returned.

These messages supplement nonzero status codes in SQLRC(2) and SQLRC(8) and provide additional diagnostic for error situations.

For errors detected by BASIC, the message contains the error code (from element 2 of status-info array) as well as message text. For a list of error codes and messages, see Appendix B, "CALL SQL Error Codes" on page 503.

Example:

```
32  Statement has an unmatched quote.
```

For errors detected by the DB2 data base system (MVS), the message contains the message id, SQLCODE, and message text. For more information, see *IBM DATABASE 2 Messages and Codes.*

Example:

```
DST408I SQLCODE = -551, ERROR: <n1> DOES NOT HAVE THE PRIVILEGE TO
PERFORM OPERATION DROP TABLE ON OBJECT <n2>

<n1> and <n2> are names supplied by the data base system.
```

For errors detected by the SQL/DS data base system (VM), the message contains only the SQLCODE, and any "message variables" returned by SQL/DS. For more information, see *SQL/Data System Messages and Codes for VM/SP.*

Example:

```
SQL/DS ERROR CONDITION, SQLCODE = -204. Parameters = <n1>, <n2>

<n1> and <n2> are names supplied by the data base system.
```

Interrupting Execution

Attention interrupts detected by BASIC when processing a CALL SQL statement are handled differently from those occurring while processing other BASIC statements. See your IBM BASIC System Services manual for information on how to cause an attention interrupt.

Normally (that is, for statements other than CALL SQL, and when an ON ATTN IGNORE or ON ATTN GOTO is not in effect), an attention interrupt causes BASIC to complete processing the current statement and then pause before executing the next. The "ATTENTION INTERRUPT AT LINE nn" message is displayed, where nn is the line number of the next statement. At this point, if you are inside the BASIC environment, you can issue immediate statements to print or set program variables. Program execution can be resumed by entering GO. See "ON Condition Statement" on page 298 and "GO Command" on page 445.

However, for attention interrupts occurring during the processing of a CALL SQL statement, you are prompted (in line mode) with the following message:

```
CALL SQL ATTN. Enter "CANCEL" to cancel the SQL request, null line to continue.
```

- If you respond CANCEL, BASIC attempts to cancel processing in the data base system. This is useful if your SQL request requires unexpected queuing or processing time and you do not care to wait. After cancel processing has completed the attention is processed normally (that is, either causes a pause or the ON ATTN actions). If you resume execution after cancel processing, unexpected results are possible because of the different data base states and SQLRC settings that can occur. See "Considerations Regarding Cancel Processing."

- If you respond with a null line (that is, just pressing the enter key), then processing of the CALL SQL statement continues. The interrupt becomes effective at the next statement, and normal attention processing occurs. Since the SQL request is not affected, resuming execution at this point does not introduce any side effects.

- If your response is anything other than CANCEL or null, then the prompt is repeated.

Considerations Regarding Cancel Processing: Because the data base system is asynchronous to BASIC, several different situations are possible when BASIC attempts to cancel your SQL request:

1. The attention interrupt is received *before* BASIC issues the SQL request to the data base system. In this case, BASIC simply does not issue the SQL request, and sets SQLRC(1) to 8 and SQLLRC(2) to 91.

2. The attention interrupt is received *after* BASIC issues the SQL request, and the CANCEL response is received *before* the data base system completes the SQL request. BASIC cancels the SQL request. This has the same effect in the data base as if a SQL ROLLBACK were performed. SQLRC(1) and SQLRC(8) are set according to the value of SQLCODE returned by the data base system.

3. The attention interrupt or the cancel response is received *after* the data base system completes the SQL request. In this case, BASIC simply ignores the cancel request and the CALL SQL statement is processed normally.

Examples

This section includes some examples illustrating the use of the CALL SQL statement. For more information and examples of how to use the CALL SQL statement, refer to *IBM BASIC Programming Guide*.

Example 1 creates (defines) a table named STOCKS. The table columns are BIN, YEAR, TYPE, STORLOC, COST, and COLOR. Example 2 inserts data (rows) into the table. The completed table is shown in Figure 30 on page 122. (*Note*: In "real life" a typical table in a relational data base would contain many more rows.)

After the table has been created and data inserted into it, Examples 3 and 4 illustrate how to issue a query to retrieve data from the table. The query is: "What is the type, location, and cost of red wines which cost more than $3.00"? The SQL statement would be:

```
SELECT TYPE,STORLOC,COST
FROM STOCKS
WHERE COST > 3.00 AND COLOR='R'
```

Examples 3 and 4 both transmit the same query to the data base system. Example 3 illustrates how a single row is retrieved with each execution of CALL SQL. Example 4 shows you how to retrieve multiple rows at a time for each CALL SQL.

Table: STOCKS

BIN	YEAR	TYPE	STORLOC	COST	COLOR
B10	1973	CHAMBERTIN	COUNTER	10.95	R
B11	1966	BORDEAUX	SPECIALTY CORNER	8.95	R
B12	1974	BORDEAUX	SPECIALTY CORNER	9.95	R
G12	1977	PORT	BASEMENT	5.88	R
C11	1971	RIOJA	IMPORT DEPARTMENT	4.95	R
C12	1971	RIOJA	WINE CLUB SHOWCASE	3.95	R
F16	1983	ROSE	WINDOW	NULL	NULL
G12	1974	MERLOT	BASEMENT	8.95	R
I10	1979	BARDOLINO	ANNEX	2.25	R
I11	1979	VALPOLICELLA	ANNEX	2.25	R
K10	1981	CHABLIS	ON ORDER	3.95	W
K11	1981	RIESLING	SHELF	6.25	W
K12	1980	SHERRY	COUNTER	6.15	NULL

Figure 30. STOCKS: A sample SQL table

Example 1— Creating a Table:

This example creates a new table in a data base. Not all users have the authority to create tables. Check with your data base administrator to see if you are authorized.

```
100 REM: Declare variables
110 OPTION BASE 1
120 INTEGER SQLRC, YEAR: DECIMAL COST
130 DIM STMT$*256, SQLRC(20), SQLMSG$*256
140 DIM BIN$*3, TYPE$*12, STORLOC$*20, COLOR$*1
150 REM: Set up the SQL statement
160 LET STMT$ = "CREATE TABLE STOCKS"
170 LET STMT$ = STMT$ & " (BIN CHAR(3) NOT NULL,"
180 LET STMT$ = STMT$ & " YEAR SMALLINT,"
190 LET STMT$ = STMT$ & " TYPE VARCHAR(12),"
200 LET STMT$ = STMT$ & " STORLOC VARCHAR(20),"
210 LET STMT$ = STMT$ & " COST DECIMAL(6,2),"
220 LET STMT$ = STMT$ & " COLOR CHAR(1)) "
230 REM: To data base system to create the table
240 CALL SQL(STMT$, SQLRC(), SQLMSG$)
250 GOSUB CHECK_SQLRC   ! Check return codes
260 REM: Issue a COMMIT to the data base
270 LET STMT$="COMMIT WORK"
280 CALL SQL(STMT$, SQLRC(), SQLMSG$)
290 GOSUB CHECK_SQLRC   ! Check return codes
300 PRINT "*** STOCKS Table Created"
310 STOP
320 REM ================================================
330 CHECK_SQLRC:        ! Subroutine to check SQLRC
340 IF SQLRC(1) LE 4 THEN
350    RETURN           ! Continue processing
360 ELSE
370    IF SQLRC(1) EQ 8  THEN
380       PRINT "*** Error detected by BASIC"
390       PRINT "    CALL SQL error code is: "; SQLRC(2)
400    ELSE
410       IF SQLRC(1) EQ 12 THEN
420          PRINT "*** Error detected by data base system"
430          PRINT "    SQL return code is: "; SQLRC(8)
440       END IF
450    END IF
460 END IF
470 PRINT "SQL statement: ";  STMT$
480 PRINT "CALL SQL message text: "; SQLMSG$
490 END
```

Note: You may want to add an "IN clause" to the SQL statement (after line 270). This clause is required for the VM environment (when using SQL/DS) and may be required or recommended by your installation for MVS (DB2). Check with your system administrator to be sure.

Example 2– Inserting rows into a table:

This example illustrates how to insert rows into a table using index values in the SQL statement. Each execution of the CALL SQL statement inserts one row. Note that in this example the data to be inserted into the table is in DATA statements within the program. There are other techniques you can use to fill a table. See *IBM BASIC Programming Guide* for examples illustrating different ways of inserting rows into a table.

```
100 OPTION BASE 1: INTEGER SQLRC, YEAR: DECIMAL COST
110 DIM STMT$*300, SQLMSG$*256, SQLRC(20)
120 DIM BIN$*3, TYPE$*12, STORLOC$*20, COLOR$*1
130 STMT$ = "INSERT INTO STOCKS " &                                    &
  &         "(BIN, YEAR, TYPE, STORLOC, COST, COLOR) " &               &
  & " VALUES (:1,  :2,   :3,   :4,      :5,   :6)"
140 DO ! Assign values to column variables and insert into table
150    READ BIN$, YEAR, TYPE$, STORLOC$, COST, COLOR$ EOF COMMIT
160    CALL SQL(STMT$, SQLRC(), SQLMSG$, BIN$, YEAR, TYPE$,     &
  &    STORLOC$, COST, COLOR$)
170    IF SQL_ERROR = 1 THEN STOP
180 LOOP
190 COMMIT: STMT$ = "COMMIT WORK"
200 CALL SQL(STMT$, SQLRC(), SQLMSG$)
210 IF SQL_ERROR = 1 THEN STOP ELSE PRINT "*** Table Completed"
220 REM ================================================================
230 DATA B10, 1973, CHAMBERTIN,   COUNTER,            10.95, R
240 DATA B11, 1966, BORDEAUX,     SPECIALITY CORNER,   8.95, R
250 DATA B12, 1974, BORDEAUX,     SPECIALITY CORNER,   9.95, R
260 DATA G12, 1977, PORT,         BASEMENT,            7.88, R
270 DATA C11, 1971, RIOJA,        IMPORT DEPARTMENT,   4.95, R
280 DATA C12, 1971, RIOJA,        WINE CLUB SHOWCASE,  3.95, R
290 DATA F16, 1983, ROSE,         WINDOW,       -2147483648, ""
300 DATA G12, 1974, MERLOT,       BASEMENT,            8.95, R
310 DATA I10, 1979, BARDOLINO,    ANNEX,               2.25, R
320 DATA I11, 1979, VALPOLICELLA, ANNEX,               2.25, R
330 DATA K10, 1981, CHABLIS,      ON ORDER,            3.95, W
340 DATA K11, 1981, RIESLING,     SHELF,               6.25, W
350 DATA K12, 1980, SHERRY,       COUNTER,             6.15, ""
360 REM ================================================================
370 DEF SQL_ERROR      ! Function to analyze SQLRC array
380 SELECT SQLRC(1)
390 CASE 0             ! Success
400    SQL_ERROR = 0
410 CASE 4             ! Warning
420    SQL_ERROR = 0
430    PRINT "*** Warning - SQLRC(8) ="; SQLRC(8)
440    PRINT "SQLRC(10-15) =";
450    FOR I=10 to 17:  PRINT SQLRC(I);: NEXT I: PRINT
460 CASE 8             ! Error - CALL BASIC
470    SQL_ERROR = 1
480    PRINT "*** Error Detected by BASIC"
490    PRINT "    CALL SQL error code is: "; SQLRC(2)
500 CASE 12            ! Error - data base
510    SQL_ERROR = 1
520    PRINT "*** Error Detected by data base system"
530    PRINT "    SQL return code is: "; SQLRC(8)
540 END SELECT
550 IF SQLRC(1) GE 4 THEN
560    PRINT "SQL statement: ";  STMT$
570    PRINT "CALL SQL message text: "; SQLMSG$
580 END IF
590 FNEND
600 END
```

Example 3— Retrieving data, a row at a time

This example illustrates a simple loop to retrieve and print selected data from the table. Each execution of the CALL SQL statement retrieves a single row and each row consists of TYPE, STORLOC, and COST for red wines costing more than $3.00.

```
100 OPTION BASE 1
110 INTEGER SQLRC: DECIMAL COST
120 DIM SQLRC(20), STMT$*1000, SQLMSG$*256
130 DIM TYPE$*12, STORLOC$*20
140 STMT$ = "SELECT TYPE,STORLOC,COST FROM STOCKS " &            &
  &         "WHERE COST > 3.00 AND COLOR='R'"
150 DO WHILE SQLRC(1) = 0
160   CALL SQL (STMT$, SQLRC(), SQLMSG$, TYPE$, STORLOC$, COST)
170   PRINT USING "<############ <################### ###.##":   &
  &   TYPE$, STORLOC$, COST
180 LOOP
190 IF SQLRC(1) = 8 AND SQLRC(2)=100 THEN
200       PRINT "*** End of Table ***"
210    ELSE
220       PRINT "Data Base or SQL Statement Error - SQLRC:"
230       FOR I = 1 TO 20:  PRINT SQLRC(I);: NEXT I: PRINT
240       PRINT "Error Message is: "; SQLMSG$
250 END IF
260 END
```

Example 4—Retrieving data, multiple rows at a time

This example is similar to Example 3; it retrieves the same data (TYPE, STORLOC, and COST of all red wines costing more than $3.00). However, rather than retrieving a single row at a time, the CALL SQL statement is set up to retrieve five rows at a time. Since seven rows in the STOCKS table satisfy the conditions of the query (red wines costing over $3.00), the first call will get five. The next call will get the remaining two and receive an end-of-table return code.

```
100 REM: Declare and initialize variables to accommodate
110 REM: the desired number of rows (in this case 5 at a time).
120 OPTION BASE 1
130 INTEGER SQLRC
140 DECIMAL COST
150 DIM STMT$*256, SQLMSG$*256
160 DIM TYPE$(5)*12, STORLOC$(5)*20, COST(5)
170 DIM SQLRC(20)
180 LET STMT$ = "SELECT TYPE, STORLOC, COST FROM STOCKS "
190 LET STMT$ = STMT$ & "WHERE COST > 3.00 AND COLOR = 'R'"
200 DO
210   CALL SQL(STMT$, SQLRC(), SQLMSG$, TYPE$(), STORLOC$(), COST())
220   EXIT IF SQLRC(1) GT 4                    ! Exit if table end
230   CALL PRINTEM (TYPE$(),STORLOC$(),COST())  ! Go print a segment
240 LOOP
250 REM: If end-of-table, determine number of rows retrieved
260 IF SQLRC(2) = 100 THEN
270    IF SQLRC(3) GT 0 THEN
280       MAT TYPE$ = TYPE$(SQLRC(3))
290       MAT STORLOC$ = STORLOC$(SQLRC(3))
300       MAT COST = COST(SQLRC(3))
310       REM: Print last (partial) segment
320       CALL PRINTEM (TYPE$(), STORLOC$(), COST())
330    END IF
340    PRINT "*** End of Table ***"
350 ELSE
360    PRINT "*** Data Base or SQL Statement Error:"
370    PRINT "     CALL SQL return code:"; SQLRC(2)
380    PRINT "    Data base return code:"; SQLRC(8)
390    PRINT "       Error message text:"; SQLMSG$
400 END IF
410 END
420 REM ============================================================
430 SUB PRINTEM (A$(), B$(), C())  ! Subprogram to print report
440 OPTION BASE 1: INTEGER I
450 PRINT NEWPAGE
460 PRINT "TYPE          STORLOC                COST"
470 PRINT
480 FOR I = 1 to SIZE(A$)
490     PRINT USING "<############ <################## ###.##":   &
  &     A$(I), B$(I), C(I)
500 NEXT I
510 END SUB
```

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 system action. See "ON Condition Statement" on page 298.

EXIT Specifies a line number or line label to which control is transferred when input or output exception conditions occur. See "EXIT Statement" on page 197.

CAUSE Explicitly causes an exception; for example, for testing purposes. See "CAUSE Statement" on page 165.

CONTINUE Resumes execution after the statement which caused an exception. See "CONTINUE Statement" on page 173.

RETRY Resumes execution after an exception by reexecuting from its beginning the statement which caused the exception. See "RETRY Statement" on page 360.

Using I/O Statement Exception-Recovery Clauses and On Condition Statements

The ON condition statement and exception-recovery 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 exception-recovery clause they are implicit; presence of a condition name implies GOTO; absence implies the action defined for the appropriate ON condition. Initially, for a program unit, the action for an ON condition is SYSTEM.

If an exception occurs, and control is transferred, whether via an exception-recovery clause or ON statement, the following intrinsic functions are available for processing the exception:

CODE Obtains the system error code

ERR Obtains the BASIC exception code

LINE Obtains the line number of the statement causing the exception

If an exception does not cause a transfer of control (the action is IGNORE or SYSTEM), the values of the above three functions are not changed.

Exception Handling in I/O Statements

Exception-recovery clauses in I/O statements take priority over the ON condition statement. When an exception occurs, BASIC first checks if an applicable exception-recovery clause is specified in the I/O statement. If it is, control is transferred as specified by the exception-recovery clause. If the exception does not correspond to any exception-recovery clause, the action taken is that which is defined for the appropriate ON condition.

The exception-recovery conditions, ENDPAGE, CONV, and SOFLOW, are equivalent to the ON conditions of the same name. All other exception-recovery clause or EXIT conditions are equivalent to the ERROR condition of the ON condition 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. See "CAUSE Statement" on page 165.

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 exception-recovery 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. See "RETRY Statement" on page 360 and "CONTINUE Statement" on page 173.

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 BASIC 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; 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 LINE 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 OFLOWLINE
110 DEF A(P1)
120   A = X/P1
130 FNEND
140 DEF B(P2)
150   ON ZDIV GOTO ZDIVLINE
160   B = P2*A(P2)
170   GOTO END_FUNC
180   ZDIVLINE:  PRINT 'ZDIV AT LINE'; LINE
190 END_FUNC: FNEND
210 X = 1.e900
220 Y = B(0)
230 Y = B(1.e-900)
240 Y = A(0)
250 STOP
260 OFLOWLINE: PRINT 'OFLOW AT LINE' ; LINE
270 CONTINUE
280 END
```

In this example, the first invocation of B (at line 220) causes an exception at line 120 in A. A is exited, and execution will resume at line 180 and LINE will be set to 160.

The next invocation of B (at line 230) will also cause an exception at line 120 in A. Both A and B will be exited; execution will resume at line 260, and LINE will be set to 230. The CONTINUE statement will transfer control to line 240. This too will cause an exception at line 120 in A. However, the ON ZDIV still points to line 180 in B, which is not active; therefore both a ZDIV exception and an "invalid exception location" exception will be generated.

Note: The invalid exception location can be handled with an ON ERROR GOTO that has a valid destination.

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.

Initially, for a program unit, the action for an ON condition is SYSTEM.

Debugging Statements

Debugging facilities let you build test points into programs. With debugging statements, you can set breakpoints, trace the execution of a program, and turn the debugging system ON and OFF. The debugging statements are ignored in compiled program units.

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; 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). See "DEBUG Statement" on page 176.

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, you can continue execution by pressing the ENTER key, or can enter 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 with the RUN command. Therefore, 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. See "BREAK Statement" on page 148. Also, see "BREAK Command" on page 423 for an alternate way to set a breakpoint without altering the program.

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.
See "TRACE Statement" on page 371.

Using the TRACE Statement

The following actions occur when tracing is on.

- For each statement causing a transfer of control (for example, a GOTO, 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 (the terminal).

Immediate Statements and Debugging

Statements that can be executed immediately (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. See "Immediate Statements" on page 133.

Immediate Statements

When you are inside the BASIC environment, you 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 "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 "BASIC Statements" on page 145.

Statements that can be used immediately are:

DEBUG
DECIMAL
DEFDBL
DEFINT
DEFSNG
DIM
INTEGER
LET
MAT
OPTION
PRINT
PRINT FIELDS
PRINT File
RANDOMIZE
REAL
REM
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.

You can continue immediate statements (except REM) 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 in Immediate Statements

When a program is stopped at a breakpoint, you can use 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, modified (has not been run since loaded, entered, or edited), or not present (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 contains a variable named BETA (but *not* a variable named ALPHA). 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 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 you wish to attach new attributes to an existing variable name. The old attributes must be detached before BASIC will accept a new declaration.

Immediate Variables with Compiled Program Units

While suspended in a compiled program unit, all variables referenced by immediate statements and the DROP command are immediate even if they have the same name as variables in the compiled program unit. Unlike source program units, where each invocation of the program unit has its own set of immediate variables, there is only one set of immediate variables for compiled program units which remains in effect until control is returned to the calling source program unit (if any), or chaining occurs. At this time, immediate variables associated with the compiled program units are dropped. When program execution terminates, the immediate variables are the ones associated with the main program.

Note: A compiled program unit may be suspended by causing an ATTN interrupt while executing the program unit with ON ATTN SYSTEM in effect or with a PAUSE statement.

Immediate Type and Dimensions

Immediate type statements (DECIMAL, DEFDBL, DEFINT, DEFSNG, INTEGER and REAL) 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 type 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 may contradict previous immediate type statements and, if at a breakpoint, program type statements. However, the new type declarations must not explicitly (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 specified in immediate type 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 does cause 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 or real type (depending on the default) as part of the DIM statement.

Immediate Statement Exceptions

All exceptions generated by immediate statements are handled according to the SYSTEM actions defined for the ON Conditions, unless program termination is specified. When program termination is specified, termination of the immediate statement occurs instead. See "ON Condition Statement" on page 298.

ON Condition statements in the workspace have no effect upon immediate statements. Also, immediate PRINT statements cannot contain exception recovery clauses (because they refer to lines in the workspace).

BASIC Commands and Editing Facilities

BASIC Commands

Abbreviation of Commands

All commands may be abbreviated, but only if the abbreviation is unique and has at least two letters. 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.

"List of Commands and their Minimum Abbreviations" on page 419 lists each command with its shortest allowable abbreviation.

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 Format 2 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:

- AUTO command
- CHANGE command
- COPY command
- DELETE command
- EXTRACT command
- FETCH command
- FIND command
- INITIALIZE command
- LIST command

MERGE command
RENUMBER command

For information on how the current line is changed, see the discussion for each command in the section "BASIC Commands" on page 419.

Specifying Files in Commands

In certain commands (such as LOAD, SAVE, and RUN), a file-spec may be used to specify the file to be used by the command. The form of the file-spec is system dependent. See your IBM BASIC System Services manual for valid entries for your system. A simple file-spec of 8 characters or less beginning with a letter and consisting of letters and digits is supported in both the VM and MVS environments.

Exclamation Mark Comments

Exclamation mark comments may follow commands. Characters following the exclamation mark are ignored. This is useful for placing comments in a log file (see "SET LOG Command" on page 479) and in a profile (see "PROFILE Command" on page 462). For example:

```
LOAD TEST1!THIS TEST IS FOR SELECT
```

When an exclamation mark appears in a command where it could be part of the command, it is interpreted as part of the command.

Example

```
RUN (PARM "*!")
```

The ! is part of the PARM string and does *not* start a comment.

```
CHANGE !A!B!ALL
```

The ! is the delimiter for old-string and new-string and does *not* start a comment.

The Workspace

IBM BASIC maintains the current program in an area called the *workspace*—an area of storage reserved for a user's exclusive use. 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. The workspace can be given a name by using one of several commands: COMPILE, FETCH, INITIALIZE, LOAD, RENAME, or RUN, or by a CHAIN statement. See your IBM BASIC System Services manual for allowed workspace names.

When you create a program it is entered, line by line, into the workspace. When you invoke BASIC, 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 it into your workspace so that those changes can be made. You get the program into your workspace from a file by using the LOAD command.

```
LOAD file-spec
```

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.

Entering Program Lines from the Terminal

IBM BASIC indicates that terminal input 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 line segment.

If you enter a continued line segment (ending with an ampersand that is not part of a REM statement), IBM BASIC prompts for a continuation line segment by displaying the leading continuation ampersand. You may respond with the continuation segment or overwrite the ampersand prompt and enter a command, immediate statement, or a program line. But if you do not respond with a continuation segment, an error will occur on the continued line.

Replacing and Deleting Lines

Program lines already entered may be replaced by reentering the line using the same line number, or deleted by entering only the line number. See "DELETE Command" on page 436 and "EXTRACT Command" on page 439 if you wish to delete groups of lines.

Full-Screen Editing

If you have a 327x type display terminal, you can use the BASIC facility for *full-screen editing*, in addition to line number editing as explained above. In essence, this editing facility works like other full-screen editing facilities; you use the cursor to change, replace, or delete characters anywhere on the screen. Although the line numbers and continuation ampersands inherent to BASIC make the editing process a little different, you will find full-screen editing to be an extremely helpful and efficient tool. For more information, see your *IBM BASIC Programming Guide*.

Editing Continuation Line Segments

A line segment within a line can be referred to by its segment number. For example, if line 200 required four segments 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 segment 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 segments as documented below. Segment numbers are never displayed.

Deleting Continuation Segments

A continuation segment may be deleted by entering:

```
line-number.segment-number
```

The segment number must be greater than or equal to 1, and must designate an existing line segment. If not, an error message is displayed and the workspace is not changed.

Example

```
110.1
```

deletes the first *continuation* segment of line 110.

Replacing Line Segments

A segment may be replaced by entering:

```
line-number.segment-number new-line-segment
```

The segment number must designate an existing segment. If not, an error message is displayed and the workspace is not changed.

If the last segment of a line is replaced *and* the new segment ends with an ampersand *and* the line is syntactically correct up to that point, the user is prompted for another continuation segment with an ampersand. This allows you to extend a line with additional segments.

Example

```
150.3&, ORGANIZATION RELATIVE &
```

replaces the third continuation segment of line 150 with

```
&,ORGANIZATION RELATIVE &
```

and prompts with an ampersand for a continuation segment.

Example

```
120.0 A = 2**B&
```

replaces the line number segment with

```
120 A = 2**B&
```

In each example, BASIC then prompts for the next line segment with an ampersand.

Inserting Continuation Segments

A new continuation segment may be inserted by entering the line number, a + sign, and the continuation segment number, as follows:

```
line-number+segment-number new-line-segment
```

The segment-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 segment number is greater than the existing number of continuation segments, then the new segment is added after the last segment in the line.

If the inserted segment is the last segment 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 segment with an ampersand.

Example

```
150+3&, TYPE NATIVE &
```

inserts a segment

```
&, TYPE NATIVE &
```

after the second continuation segment of line 150 and prompts with an ampersand. The inserted segment is then the third continuation segment.

Part Two: Statements, Functions, and Commands

This section describes the syntax and semantics of each statement, function, and command in BASIC. The statements are listed first, followed by the functions and the commands. They are listed alphabetically within each category.

Syntax Notation

This manual uses the following conventions for syntax notation in the function, command and statement descriptions:

Symbol	Meaning

[ ]	Brackets enclose optional data that can be omitted without causing errors.
	
	Example The RUN command:
	
	`RUN [program-name]`
	
	shows that both forms (below) of the RUN command are valid:
	
	`RUN`
	
	`RUN program-name`

|	used in syntax notation, the | sign separates alternative items. You select only one of them. *Example* `DISPLAY|INTERNAL|NATIVE` You choose one of the alternatives: DISPLAY, INTERNAL, or NATIVE.

{ } Braces indicate a choice of required operands. Choose one alternative enclosed in the braces.

Example

```
{DISPLAY|INTERNAL|NATIVE}
```

You *must* choose one and 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.

BASIC Statements

LET (Scalar Assignment) Statement
LINE INPUT/LINPUT Statement
LINE INPUT/LINPUT File Statement
LOOP Statement
MARGIN Statement
MARGIN File Statement
MAT (Array Assignment) Statement
NEXT Statement
ON Condition Statement
ON GOTO/GOSUB Statement
OPEN Statement
OPTION Statement
PAUSE Statement
PRINT Statement
PRINT FIELDS Statement
PRINT File Statement
RANDOMIZE Statement
READ Statement
READ File Statement
REAL Statement
REM Statement
REREAD Statement
RESET Statement
RESTORE Statement
RETRY Statement
RETURN Statement
REWRITE Statement
SCRATCH Statement
SELECT Statement
STOP Statement
SUB Statement
SUBEXIT Statement
TRACE Statement
USE Statement
WRITE Statement

Descriptions of BASIC Statements

This section describes the syntax and semantics of each statement in the BASIC language. The statements appear in alphabetic order. For the relationships between statements, see "BASIC Statements" on page 71.

For each statement, you'll find:

- A brief description
- Statement format (see "Syntax Notation" on page 143)
- Explanation of items in the format
- A full description
- Error conditions, if any (Errors occur because of incorrect syntax or semantics and are detected before execution is started.)
- Exception conditions, if any (Exceptions occur when the program is executing.)
- Immediate execution description if this statement can be executed in immediate mode

BREAK Statement

The BREAK statement suspends program execution.

Format

```
BREAK
```

Description

The BREAK statement provides a means of using the debugging capabilities of IBM BASIC.

When debugging is active, execution of a BREAK statement suspends program execution, identifies the current line, and makes possible interaction with the system. The suspension of execution is known as a breakpoint.

The BREAK statement is controlled by the DEBUG statement. See "Debugging Statements" on page 130 and "DEBUG Statement" on page 176.

Execution of a BREAK statement when debugging is inactive has no effect.

When a program is suspended at a breakpoint, you can use commands. If you do not modify the program, you can resume execution either with a GO command or a null entry. (A null entry can be used only if no other commands have been entered while at the breakpoint.)

If you modify the program 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, INITIALIZE, LOAD, RENUMBER, and RUN.

A BREAK statement causes STEP mode to be terminated (see "RUN Command" on page 472). To continue execution in STEP mode, you must enter the GO command with the STEP option.

See "BREAK Command" on page 423 for another method of causing a program break (without the necessity of editing and rerunning your program).

BREAK statements in compiled program units are ignored.

See "DEBUG Statement" on page 176 for an example of how a BREAK statement is used with TRACE and DEBUG statements.

CALL Statement

The CALL statement invokes a subprogram.

Format

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

where:

name
identifies the subprogram to be run. Subprogram names may contain at most seven characters. (See "Identifiers" on page 8 for more information.)

If the name is BASIC, CLINK, COBOL, FLINK, FORTRAN, GDDM, PLI, PLINK, SQL, or SYSTEM, see the special format on the following pages.

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

There may be from 0 to 6 commas to indicate from 1 to 7 dimensions. The number of commas is not significant except that it may be used to implicitly declare the array in the calling program unit.

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 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, type and precision of arguments used in the CALL statement must agree with the number, type, and precision of parameters in the corresponding SUB statement. Numeric arguments will *not* be converted.

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 unit.
- Any assignment to such a parameter in a subprogram is an assignment to the corresponding argument in the calling program unit.

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 resulting value is assigned to a temporary location available only to the subprogram. In any reference to the corresponding subprogram parameter, the value in this temporary location is used; any assignment to the corresponding subprogram parameter is an assignment to this temporary location.

See also "Subprogram Statements" on page 93, "SUB Statement" on page 368, "END SUB Statement" on page 196, and "SUBEXIT Statement" on page 370.

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 Display Manager (GDDM), to execute the SET TERM command (CALL BASIC), to request operations from IBM DATABASE 2 (DB2) or SQL/DS, or to execute host operating system commands (CALL SYSTEM). This is done by specifying special subprogram names in CALL statements. (See "CALL BASIC Statement" on page 151, "CALL SYSTEM Statement" on page 161, "CALL GDDM Statement" on page 155, "CALL COBOL, FORTRAN or PLI Statement" on page 152, and "CALL SQL Statement" on page 159.) The special subprogram names are:

```
BASIC       GDDM
CLINK       PLI
COBOL       PLINK
FLINK       SQL
FORTRAN     SYSTEM
```

These names are keywords and also reserved words.

If your organization has removed these names from the reserved word list, you can use them as variable, array or function 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 BASIC Statement

The CALL BASIC statement enables you to execute the SET TERM command in a BASIC program, which allows you to choose line or scroll mode.

Format

```
CALL BASIC (set-term-string)
```

where:

set-term-string
is a character expression which evaluates to a valid SET TERM command, that is, SET TERM SCROLL or SET TERM LINE.

Description

CALL BASIC is used for the program statement equivalent of the SET TERM command. SET TERM SCROLL or LINE in a CALL BASIC statement affects the subsequent terminal input and output of the program it is invoked from, as well as input and output inside the BASIC environment.

Example

```
* SET TERM LINE
* 100 CALL BASIC ("SET TERM SCROLL")
* 110 FOR X = 1 TO 3
* 120   PRINT X
* 130 NEXT X
* 140 END
```

Line mode is in effect when the program begins to execute. The CALL BASIC statement changes the mode to SCROLL, and the program output will be produced in SCROLL mode. When the program finishes executing, terminal input and output will continue to be in SCROLL mode.

CALL COBOL, FORTRAN or PLI Statement

The CALL COBOL, FORTRAN, or PLI statements link to routines written in COBOL, FORTRAN, or PL/I. These are called *interlanguage calls* (ILC).

Format

```
CALL {COBOL|FORTRAN|PLI} (program-name [,argument]...)

CALL {CLINK|FLINK|PLINK} [(program-name [,program-name]...)]
```

where:

program-name
is a character expression which is the name of a routine to be called.

argument
is an argument that is converted, as needed, as shown in Figure 31 on page 153.

Description

Before your BASIC program can call COBOL, FORTRAN, or PL/I programs, BASIC may require that it be told which programs are to be called in a CALL CLINK, CALL FLINK, or CALL PLINK statement, respectively. Your IBM BASIC System Services manual gives details.

Because of differences in internal data representations, the called programs must return values to BASIC carefully. Only integer, real, and character arguments may be returned to BASIC, and they must conform to the characteristics of BASIC (fullword integers, real single, or real double, and character strings of the proper length). Decimal values cannot be returned by the called program. Character variables and array elements (without substring qualifiers) are passed by reference (see "CALL Statement" on page 149).

Integer and real arguments are passed to the called routine without conversion. Character arguments are converted as described below. Decimal arguments are converted to real double, but decimal arrays cannot be used as arguments in interlanguage calls.

BASIC	COBOL	FORTRAN	PL/I
INTEGER	PIC S9(9) USAGE COMP-4	INTEGER	FIXED BIN(31)
REAL SINGLE	USAGE COMP-1	REAL*4	FLOAT BIN(21)
REAL DOUBLE	USAGE COMP-2	REAL*8	FLOAT BIN(53)
DECIMAL [1]	USAGE COMP-2	REAL*8	FLOAT BIN(53)
CHARACTER [1]	PIC X(n) USAGE DISPLAY	CHARACTER*(*) CHARACTER*n	CHAR(n) VARYING CHAR(*) VARYING

Figure 31. Type Equivalences for Interlanguage Calls

Notes:

1 Conversion required.

In PL/I, FLOAT BIN(53) is equivalent to FLOAT DECIMAL(15).

Interlanguage Calls with Scalar Character Arguments

When calling COBOL, the current length (at the time of the call) of a character argument must be equal to n in Figure 31.

When calling PL/I, the current length must be less than or equal to n in Figure 31; and, if a value is to be returned, the maximum length defined for the BASIC variable or array element must be equal to n. If an asterisk (*) is used in the PL/I declaration, PL/I uses the current length and maximum length passed by BASIC.

When calling FORTRAN, the current length of a character argument must be equal to n in Figure 31 if the corresponding parameter is declared as CHARACTER*n. If the corresponding parameter is declared as CHARACTER*(*), FORTRAN uses the current length passed by BASIC. For character arguments the current length of the value must not be zero. If it is zero, an exception (parameter mismatch) will occur.

Interlanguage Calls with Array Arguments

Arrays are passed to COBOL, FORTRAN, and PL/I using the empty declarator syntax as defined for a CALL statement. As explained earlier, decimal arrays may not be passed as arguments. Instead, use a MAT statement to convert to an appropriate type and use that array as the argument. If a decimal array is passed, an exception (parameter mismatch) will occur.

Values may be returned in elements of an array.

Integer, real single, and real double arrays are passed by reference.

Character arrays are passed by value-return, as explained in the following paragraphs.

For COBOL and FORTRAN, the character elements of the temporary array are fixed length and have a length equal to the maximum length defined for the array elements. Each element in the temporary array is padded with spaces to the maximum in the temporary array. On return, the values of the temporary array (up to the corresponding element's current length) are copied back into the argument array. The current length will be unchanged on return. Therefore, any characters that are after the current length (which may be zero) which were changed by the FORTRAN subroutine or COBOL subprogram will not be accessible by the BASIC program unit. It is up to the user to make sure all current lengths are set as desired before the call.

When calling COBOL, the maximum length of a character array must be equal to n in Figure 31 on page 153. When calling FORTRAN, the maximum length must be equal to n in Figure 31 on page 153 if the corresponding parameter is declared as CHARACTER*n. If the corresponding parameter is declared as CHARACTER*(*), FORTRAN uses the maximum length passed by BASIC.

RPAD$ may be used in a MAT statement to pad with spaces all the elements of an array to the desired length (get current length up to max). See "Scalar Function References" on page 284.

Example

```
10 DIM A$(30)*(20)
20 MAT A$ = RPAD$(A$,20)
```

For PL/I, the character elements of the temporary array are unaligned and varying. The maximum length defined for the BASIC array must be equal to the length n (see Figure 31 on page 153) defined in PL/I, whether or not values are returned. If the length in PL/I is defined as an asterisk (*), PL/I uses the maximum length passed by BASIC. The current length of each element on return is updated to the current length as returned for the corresponding element of the temporary array.

NOTE: BASIC stores arrays with the rightmost subscript varying most rapidly while FORTRAN stores arrays with the leftmost subscript varying most rapidly. When calling FORTRAN, BASIC does not rearrange an array argument to be stored in FORTRAN order. If it is desired to have the array stored in FORTRAN order, the user should use the TRN array function in a MAT statement before the call. After the call, the array may be transposed back if desired. See "Transpose Array Function (TRN)" on page 294 for additional information.

See also "Calling Subprograms Written in Other Languages" on page 96.

CALL GDDM Statement

The CALL GDDM statement performs graphic functions within an IBM BASIC program using the Graphical Data Display Manager (GDDM).

Format

```
CALL  GDDM (function [,argument]...)
```

where:

function
is either

a character expression whose value is the name of a GDDM or presentation graphics feature (PGF) function. Leading and trailing spaces in the value are ignored.

or

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 *Graphical Data Display Manager User's Guide*.

argument
is an actual argument for the operation. The number and type of these arguments depends upon the function.

Description

You may specify a requested function by name or by a request control parameter (RCP) code.

The Call Format Descriptor Module (described in the *Graphical Data Display Manager Base Programming Reference*) is used in determining the appropriate RCP code associated with the name given in a CALL GDDM statement and for the types of arguments.

The arguments will be checked for validity to the extent indicated by the Call Format Descriptor Module.

Numeric arguments are converted to real single or integer as indicated by the Call Format Descriptor Module.

GDDM does not use BASIC's varying length character data types. BASIC converts character arguments to fixed length character strings (the fixed length is equal to current length).

You cannot pass character arrays. A character value must be used to correspond to character array parameters.

For more information, refer to your IBM BASIC System Services manual and "Calling the Graphical Data Display Manager (GDDM)" on page 100.

Example

```
100 !    This BASIC program uses GDDM
110 !     to draw a solid red triangle
120 CALL GDDM ("GSCLR")                        ! Clear the Graphics Field
130 CALL GDDM ("GSPS",.98,1.0)                 ! Set-up Picture Size
140 CALL GDDM ("GSWIN",0.0,100.0,0.0,100.0)    ! Coordinates Range
150 CALL GDDM ("GSPAT",16)                     ! Pattern --> Solid
160 CALL GDDM ("GSCOL",2)                      ! Color --> RED
170 CALL GDDM ("GSAREA",1)                     ! Start Area for Triangle
180 CALL GDDM ("GSMOVE",30.0,30.0)             ! Starting Point
190 CALL GDDM ("GSLINE",50.0,70.0)             ! Draw Left Side
200 CALL GDDM ("GSLINE",70.0,30.0)             ! Draw Right Side
210 CALL GDDM ("GSLINE",30.0,30.0)             ! Draw Bottom
220 CALL GDDM ("GSENDA")                       ! ...and Send to Page
230 CALL GDDM ("ASREAD",attype,atmod,count)    ! Send Page to Terminal
240 CALL GDDM ("GSCLR")                        ! Clear the Graphics Field
250 CALL GDDM ("CHTERM")                       ! Terminate PG Routines
260 END                                        ! End IBM BASIC Program
```

Interactive Chart Utility

The Interactive Chart Utility (ICU) may be called from a BASIC program using a CALL GDDM statement specifying the CHART function.

The Interactive Chart Utility and the Chart function is discussed in *Graphical Data Display Manager Presentation Graphics Feature: Programming Reference* and *Graphical Data Display Manager Presentation Graphics Feature: Interactive Chart Utility User's Guide.*

The arguments for the CALL GDDM statement when specifying the CHART function are as follows:

Format

```
CALL  GDDM (chart-expression, a(), b(), c$, d(), x(), y(), k$, l$, h$)
```

where:

chart-expression
is a character expression whose value is CHART (leading and trailing spaces are ignored), or a numeric expression whose rounded integer value is the request control parameter (RCP) for the CHART function (335544320).

a(), b(), c$, d(), x(), y(), k$, l$, h$
are the arguments for the CALL GDDM statement when specifying the CHART function. The arguments correspond to the CHART parameters as shown in the following:

a is an integer array:

- **a(0)** is LEVEL : 0 or 1
- **a(1)** is DISPLAY : 0 to 8 (8 invalid with LEVEL 0)
- **a(2)** is HELP : 0 or 1
- **a(3)** is ISOLATE : 0 to 2
- **a(4)** is BINDING : 0 or 1
- **a(5)** is NG : 0 to 999
- **a(6)** is NE : -999 to 999
- **a(7)** is KEYL : 0 to 132
- **a(8)** is LABELL : 0 to 132
- **a(9)** is HEADINGL : 0 to 132
- **a(10)** is PRTDEP (if LEVEL is 0)
- **a(11)** is PRTWID (if LEVEL is 0)
- **a(12)** is PRTCOPY : 0 to 99
- **a(13)** is PRTHEAD : 0 to 2 (if LEVEL is 1)
- **a(14)** is PRTUNIT (if LEVEL is 1)
- **a(15)** is reserved : 0 (if LEVEL is 1)
- **a(16)** is DRYTYPE : 0 to 5 (if LEVEL is 1)
- **a(17)** is DRYTYPEQ : 0 to 3 (if LEVEL is 1)
- **a(18)** is EXPLVL : 0 to 2 (if LEVEL is 1)

b is a real single array:

- **b(0)** is PRTDEP (if LEVEL is 1)
- **b(1)** is PRTWID (if LEVEL is 1)
- **b(2)** is PRTVOFF (if LEVEL is 1)
- **b(3)** is PRTHOFF (if LEVEL is 1)

c$ is a character string:

c$(1:8)	is	FORMNAME
c$(9:16)	is	DATANAME
c$(17:24)	is	PRTNAME
c$(25:32)	is	DRYNAME (if LEVEL is 1)
c$(33:40)	is	DRYLIB (if LEVEL is 1)

d	is an integer array corresponding to	DATA CONTROL
x	is a real single array corresponding to	X
y	is a real single array corresponding to	Y
k$	is a character string corresponding to	KEYS
l$	is a character string corresponding to	LABELS
h$	is a character string corresponding to	HEADINGS

For all the above arrays, the subscripts are shown with the BASE 0 option in effect. If the BASE 1 option is in effect, add one to all the subscripts shown above. An exception (parameter mismatch) will occur if an array has more than one dimension or has the wrong type.

A CALL GDDM statement for the CHART function builds the GDDM parameter CHART_CONTROL from arrays a, b, and c$.

See *Graphical Data Display Manager Presentation Graphics Feature: Programming Reference* for more details and restrictions on the CHART parameter values and their meanings. See also the GDDM examples in Appendix B of your *IBM BASIC Programming Guide*.

The GDDM parameters KEYS and LABELS are defined as character arrays in the above GDDM manual but in the CALL GDDM statement, k$ and l$ are character strings. The substring from (i*length+1) to (i+1)*length corresponds to the *ith* element in the array (where 0 is the first subscript and length is the length of an element). If a label or key is less than the appropriate length, it should be padded with spaces to that length in the character substring.

The arrays d, x, and y must contain at least the appropriate number of elements as required by the chart control information (defined by a, b, and c$). The character strings that are the values of k$ and l$ must have at least the appropriate number of substrings specifying the keys and labels as required by the chart control information. Any extra array elements or character substrings are ignored.

Incorrect values for arguments may cause the Interactive Chart Utility to fail which in turn may cause severe or possibly unrecoverable errors to occur in BASIC.

CALL SQL Statement

The CALL SQL statement accesses or manipulates data stored in a relational data base.

Format

```
CALL SQL (SQL-stmt, status-info(), message-var [,value-list])
```

where:

SQL-stmt
is a character expression, the value of which is an SQL statement. The SQL statement is a variable-length character string with a maximum length of 32765. The SQL statement may be entered in upper or lowercase. If entered in lowercase all characters not enclosed in apostrophes or quotation marks within the statement will be interpreted as uppercase by the data base system.

status-info
is the name of an integer array in which SQL information and return codes are returned when processing of the SQL statement is complete. This array must be an integer array with at least 20 elements. The information and codes returned in this array after the CALL SQL is executed come from both BASIC and the data base system.

Certain elements of the array are sometimes used to convey information to the data base system during execution of the CALL SQL statement (values are assigned to these elements by your BASIC program prior to execution of the CALL SQL statement).

The meaning and use of each element of the status-info array are discussed in "The status-info Array" on page 114.

If BASIC detects an error in the first three arguments (for example, the *status-info* argument is not an integer array of at least 20 elements), it generates an exception condition which can be tested with the ON Condition statement.

message-var
is a character variable into which messages are stored when processing of the SQL statement is complete. The maximum length of a message is 256 characters. It may represent either a BASIC or a data base error (see "SQL Messages" on page 120).

value-list
consists of one or more constants, variables (scalar or array) or expressions, as appropriate. However, for data retrieval, the values may only be variables or arrays. Value arguments are required for any SQL statement which produces a result table or which contains index values used to transmit information to the data base. (Index values are discussed in greater detail in the section "Index Values in SQL Statements" on page 103.)

CALL SQL Statement

Description

The data base access request is specified in the Structured Query Language (SQL). The data base management system which processes the SQL request depends on the operating system your BASIC program is running under:

- IBM DATABASE 2 (DB2) for MVS
- Structured Query Language/Data System (SQL/DS) for VM

For more information and examples of how the CALL SQL statement is used to access relational data bases, see "Using CALL SQL to Access Relational Data Bases" on page 100.

CALL SYSTEM Statement

The CALL SYSTEM statement allows you to execute a limited set of system commands.

Format

```
CALL SYSTEM (character-expression)
```

where:

character-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. For a list of the commands available, see your IBM BASIC System Services manual. Be careful when using these commands; some of them can adversely affect your BASIC terminal session or your program's execution.

When the CALL SYSTEM statement is executed, the system command in the *character-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. For methods of restoring the BASIC screen, see your IBM BASIC System Services manual.

If an error occurs during command execution, an exception occurs.

See also "Calling the System" on page 96, and "SYSTEM Command" on page 491.

CASE Statement

CASE Statement

The CASE statement immediately precedes a group of statements (the CASE block statements) 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.

See "SELECT Statement" on page 366, "CASE ELSE Statement" on page 164, and "END SELECT Statement" on page 195. See Figure 11 on page 56 for a list of the relational operators.

CASE blocks include all statement lines between a CASE statement and either the next CASE statement, a 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 79.

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.

Example

```
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 within a SELECT block that are executed if none of the CASE blocks are selected. The group of statements is referred to as a CASE ELSE block.

Format

```
CASE ELSE
```

Description

The CASE ELSE statement defines the CASE ELSE block to be executed if none of the selection criteria for the CASE blocks are met.

The CASE ELSE statement is used with the SELECT, CASE, and END SELECT statements to form SELECT blocks. These are discussed under "SELECT Blocks" on page 79.

The CASE ELSE statement may only appear within a SELECT block. If present, it must be placed after the last CASE block and prior to the END SELECT statement for the SELECT block.

See also "SELECT Statement" on page 366, "END SELECT Statement" on page 195, and "CASE Statement" on page 162.

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 or real, rounded and converted to integer. The result is used as the exception code.

Exception codes are listed in Appendix A, "Exception Codes" on page 493. 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 298).

If a CAUSE statement causes the message for an exception to be displayed, and that exception requires the substitution of a value, the value will appear as &1 in the message.

See also "Exception Handling Statements" on page 127.

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, using the ERR intrinsic function (see "ERR Exception Code" on page 391) and, if so, 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.

Note that if line 510 had not set ON ERROR SYSTEM, the program would have gone into an infinite loop.

CHAIN Statement

The CHAIN statement halts processing of the program in which it appears and starts a new main program.

Format

```
CHAIN character-expression [,FILES] [,name]...
```

where:

character-expression
is a character expression whose value is a file specification for the chained program (the name of the file containing the new main program being started). See your IBM BASIC System Services manual for allowed file specifications.

name
is a simple variable or an array name.

Note: Array names in a CHAIN statement, unlike those in a CALL statement, must not be stated as empty array declarations. Just use the names.

Description

Chaining allows separate programs to be processed serially, without outside intervention. This procedure is useful when segmenting large programs (when breaking them into smaller, more manageable pieces).

When a CHAIN statement is executed, the program in process is terminated and the main program in the program named in the CHAIN statement is invoked.

When you CHAIN to another program, some or all of the memory occupied by the chaining program is usually available for use by the chained program. The exceptions are:

- The memory used by Library routines is not available, nor is the memory occupied by dynamically loaded subprograms which were not part of a dynamically loaded main program. Those routines and subprograms are still available for use by the chained program.

- If your chaining program has called GDDM, SQL, or another language program unit (COBOL, FORTRAN, PL/I), the memory used by the called routines is still allocated for use by GDDM, SQL, or the other language program unit. For calls to dynamically loaded COBOL, FORTRAN, or PL/I program units, you can make the memory available by calling CLINK, FLINK, or PLINK, respectively, in either the chaining program (before the CHAIN is executed), or in the chained program.

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.

Name: Each name defines the name of a variable or array in the chaining program that is to retain its current values when the chained main program begins executing. All other data except COMMON variables are destroyed during the chaining operation. *Name* may not be the name of a variable or array in COMMON. If the CHAIN statement is specified in a subprogram, *name* may not be the name of a parameter.

The passed attributes for each variable or array name in the USE statement are matched against the locally defined attributes for the same variable or array name. An attribute mismatch results in an exception. Note that the passed attributes are not automatically inherited by the chained main program; each variable or array name in the USE statement must be defined within the chained main program to have the appropriate attributes.

If a name matches, but the type, maximum number of elements for an array, or the maximum character length for a character variable or array do not match, then an exception occurs in the chained main program (since the ON action is SYSTEM at entry to a program unit, the program will terminate).

An array in the chaining program unit must be defined (with a DIM or by default) by some other statement within the chaining program unit.

A CHAIN statement may not be used in a function definition (between DEF and FNEND statements). It may occur in both main programs and subprograms.

See your IBM BASIC System Services manual for allowed values of the chained program name and the order of precedence as to whether a source program or compiled program is loaded.

See also "USE Statement" on page 373 and "Chaining Statements" on page 96.

Example

Program X contains this statement

```
100 CHAIN "LINK2",FILES,C,B,A
```

Program LINK2 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 LINK2. The values of variables A, B, and C are passed to LINK2 from X. The variables A, B, and C must be the same type in both program X and program LINK2. For example, if variable A is typed integer in program X and decimal in program LINK2, an exception occurs.

Note: Names do not have to be listed in the same order.

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 an integer within the range 1 to 255.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
EOF line-reference
IOERR line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

Description

The CLOSE statement prevents further 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 type file which has an incomplete print line waiting to be written (the last PRINT statement ended with a comma or semicolon) causes the print line to be written before the file is closed. For more information, see "BASIC File Capabilities" on page 61.

The STOP and END statements, and CHAIN statement without the keyword FILES, automatically close all active files. Any resulting exceptions 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 terminal, causes an exception.

Exception Conditions: The two exception conditions EOF (end-of-file) and IOERR (input/output error) may be recoverable if an exception-recovery 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 exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 for the file, control passes to line number 800.

COMMON Statement

The COMMON statement provides a means of sharing values between program units of a program or between programs when a CHAIN statement is executed.

Format

```
COM[MON] {numeric-com | char-com[*length]} [,{numeric-com | char-com[*length]}]...
```

where:

numeric-com
is a numeric variable name or numeric array-declarator.

char-com
is a character variable name or character array-declarator.

length
is an unsigned integer constant between 1 and 32767, which defines the maximum string length.

and where:

array-declarator
has the form:

```
array-name (d1[,d2[,d3[,d4[,d5[,d6[,d7]]]]]])
```

where:

array-name
is a numeric or character identifier.

d
Each *d* is an unsigned integer constant in the range 0 or 1 (depending upon the BASE option) to 32767. It defines the upper bound of that dimension. See "OPTION Statement" on page 312, and "DIM Statement" on page 188.

Description

The COMMON statement explicitly defines the number and size 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 a 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.

As each program unit using COMMON is executed, BASIC checks its common area against the established common area. All of a program's units (main programs and subprograms) must agree in their definition of the common area up to the extent of the smaller. 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. Corresponding items are checked according to the following rules:

- A scalar must correspond to a scalar.
- An array must correspond to an array.
- Type has to be the same.
- Character data must have the same maximum length.
- Total size of arrays must be the same.

Arrays can be declared with different dimensions, provided the total size remains the same. Sizes are checked when a common array is shared between program units, but the array's dimensions remain as they existed in the calling program. If a called subprogram redimensions a common array, the new dimensions are retained when control is returned to the caller.

The following examples illustrate correct and incorrect usage of COMMON.

Example (correct)

```
100 COM A,B%(3,3),C$(16)*10
110 COM D(5)
```

creates a common area (assuming OPTION ARITHMETIC DECIMAL and BASE 0) 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, and a 6-element, one-dimensional decimal array named D.

Example (incorrect)

```
100 COM A
110 A(3) = PI
```

causes an exception because common arrays must be explicitly dimensioned in the COMMON statement.

Example (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 (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%
```

Example (incorrect)

```
10 OPTION SPREC
15 REAL A
20 COM A
30 CALL JOE
40 END
50 SUB JOE
60 OPTION LPREC
65 REAL X
70 COM X
80 END SUB
```

contains an error: subprogram JOE's image of common differs from that of the main program because A and X have different precision (A is real single and X is real double).

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.

If an exception condition does not exist when CONTINUE is executed, an exception occurs.

See also "Exception Handling Statements" on page 127, "ON Condition Statement" on page 298, and "RETRY Statement" on page 360.

Example

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 exception, just count the occurrences.

```
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.

DATA Statement

The DATA statement is used to create internal data tables for reference by READ statements.

Format

```
DATA [integer*]item [,[integer*]item]...
```

where:

integer
is a nonzero, unsigned integer constant used to replicate the following item.

item
is a constant (either numeric or character) or an unquoted character string.

Description

DATA statements are nonexecutable statements which are used to create a data table internal to the program unit. They can appear anywhere in the program unit, but all of the DATA statements create one internal data table of values whose order is determined by the line numbers of the statements.

A DATA statement must be the last statement on a line. In addition, if the DATA statement is referenced by line number or line label in a RESTORE statement, it must be the first (and therefore only) statement on a line.

Both character and numeric constants can be used in DATA statements. The following rules apply to their use.

Character constants: A character constant may be specified without surrounding quotation marks provided:

- The character constant does not start with a nonzero, unsigned integer followed by an asterisk.
- The character constant contains no commas, leading or trailing spaces, and no leading or trailing quotation marks.
- The last character constant in the DATA statement does not end with the ampersand (&) character.
- The character constant is not a null string.

Numeric constants: A valid numeric constant may be used for assignment to either a numeric or a character variable:

- If it is assigned to a numeric variable, it is assigned as a numeric value.
- If it is assigned to a character variable, it is assigned as a string of characters.

Specifying a replication factor (for example, *10*MOB*) is equivalent to repeating the unquoted character string *MOB* 10 consecutive times.

See also "READ Statement" on page 344, "RESTORE Statement" on page 358, and "Data Table Statements" on page 87.

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 table 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

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 the trace listing.

Description

BASIC provides debugging facilities to allow you to build test points into your 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 subsequent 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.

If debugging is active and tracing is active, execution of a DEBUG ON statement will turn tracing off. A TRACE ON statement must be executed again to activate tracing.

The fileref in the optional TO clause overrides any subsequent TO clauses in TRACE statements. If you specify a fileref other than 0, that fileref must designate a file that was opened as OUTPUT, DISPLAY, and SEQUENTIAL (see "OPEN Statement" on page 305).

Debugging normally takes place during program development when the source program is being run interactively. Therefore, the debugging facilities do not apply to compiled (object) program units:

- DEBUG statements are ignored.
- Neither program flow nor variable assignments will be shown.
- BREAK statements will be ignored.
- Therefore, debugging statements may be left in the source code to aid later maintenance, but will have no effect on production runs using the compiled program.

See "Debugging Statements" on page 130, "TRACE Statement" on page 371, and "BREAK Statement" on page 148.

The following example illustrates the use of the DEBUG statement in conjunction with the TRACE and BREAK statements:

Example

```
 * LIST
100 DEBUG ON
110 TRACE ON
120 LET Y = 100
130   FOR X = 1 TO 3
140     Z = X * 2
150   NEXT X
160 TRACE OFF
170 LET SUM1 = X + Y + Z
180 BREAK
190 DEBUG OFF
200 END
 * RUN
TRACE ASSIGNMENT AT LINE 120 VALUE:  100
TRACE ASSIGNMENT AT LINE 130 VALUE:  1
TRACE ASSIGNMENT AT LINE 140 VALUE:  2
TRACE ASSIGNMENT AT LINE 150 VALUE:  2
TRACE BRANCH FROM LINE 150 TO LINE  130
TRACE ASSIGNMENT AT LINE 140 VALUE:  4
TRACE ASSIGNMENT AT LINE 150 VALUE:  3
TRACE BRANCH FROM LINE 150 TO LINE  130
TRACE ASSIGNMENT AT LINE 140 VALUE:  6
TRACE ASSIGNMENT AT LINE 150 VALUE:  4
BREAK STATEMENT AT LINE 180.
 * PRINT SUM1
 110
 * GO
END AT LINE 200.
```

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.

The immediate DEBUG statement does not apply to compiled program units (it will be ignored).

See also "Immediate Statements" on page 133.

DECIMAL Statement

The DECIMAL statement specifies which identifiers are to be assigned decimal type.

Format

```
DECIMAL [{identifier|(letter-list)} [,{identifier|(letter-list)}]...]
```

where:

identifier
is a 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.

Description

The DECIMAL statement declares a specific identifier or any identifier beginning with a specific letter as having decimal type. 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 also the default if the ARITHMETIC DECIMAL option is in effect (see "OPTION Statement" on page 312).

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 user-defined function names.

In a DECIMAL statement, an identifier may end with the self-typing character #, only when the ARITHMETIC DECIMAL option is in effect, but not with % or $. If a DECIMAL statement specifies a *letter-list*, 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 another typing statement (DEFSNG, DEFDBL, DEFINT, INTEGER, or REAL) by identifier. The letter list may be specified as either single letters, such as (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.

See "Variables" on page 31 and "Arrays" on page 36 for more information on typing.

Example

```
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.

COST# would be decimal or real, depending on whether the ARITHMETIC DECIMAL or ARITHMETIC NATIVE option is in effect.

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 133 and "Immediate Type and Dimensions" on page 135 for the rules regarding the interaction with other immediate statements and program statements.

DEF Statement

The DEF statement defines and names a user-defined function. The DEF statement can define either a single-statement or multi-statement function. The FNEND statement indicates the end of a multi-statement function.

Format

```
DEF name [(parameter [,parameter]...)] [=expression]
```

where:

name
is a scalar numeric or character name that gives a name to the function. If it is character (ending 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). If the length is not specified, the maximum defaults to a value determined by your system administrator. The IBM-distributed default is 18.

parameter
is a scalar numeric or character variable name that specifies the function input. If it is character (ending with $), the maximum length may be defined by following the identifier with an asterisk and an integer. If the length is not specified, the maximum defaults to a value determined by your system administrator. The IBM-distributed default is 18.

expression
is an expression of the same type, numeric or character, as *name,* used to complete a single-statement 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. The result type and the type of the parameters is determined as for variables. See "Variables" on page 31 for more information.

A user-defined function is referred to in other statements within the same program unit in a manner similar to the intrinsic functions (see "Functions" on page 43). 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 identifier X, enclosed in parentheses after the function name, is called a parameter. You can have more than one parameter, and each may be either a numeric or character variable. The function performs its defined calculation on the actual values supplied for these parameters by the corresponding arguments used in the function reference.

When a user-defined function is referenced (an expression involving the function is evaluated), any arguments in the function reference are evaluated and their values are assigned (passed by value) to the parameters of the DEF statement. Therefore, values assigned to function parameters within the function have no effect on the corresponding arguments in the invoking function reference. Numeric arguments are converted as needed to the type of a corresponding numeric parameter.

Example

```
100 DEF E_TO_X_SQUARED(X) = EXP(X**2)
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 out of user-defined functions, other than through function references, CALL statements, and exceptions, is not allowed. For exception handling, see "ON Condition Statement" on page 298.

Nonexecutable statements, such as COMMON, DATA, DECIMAL, DIM, EXIT, FORM, IMAGE, OPTION, and USE may appear within a user-defined function and affect the entire program unit. DATA, EXIT, FORM, and IMAGE statements may be referred to from anywhere in the program unit.

A parameter identifier appearing in the parameter list of a function definition is local to that function definition; it is distinct from identifiers with the same name outside the function definition.

A user-defined function can refer to or change values of any variable outside the function definition except those used as parameters in the function, in the program unit containing the function. A user-defined function may also change the value of a parameter, but this does not affect the corresponding argument nor a variable with the same name as the parameter that is outside the function.

Undefined results may occur if a user-defined function changes the value of a variable appearing in the same statement as the function reference.

If a user-defined function performs any input/output, and the function was referenced directly or indirectly from an input/output list, the referencing input/output statement will cause an exception if control is returned to it.

A function of a given 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. Recursive function invocations are not permitted.

Single-Statement Functions: If a function is completely defined in a DEF statement, the expression in that statement is evaluated and its value assigned as the value of the function when the function is referenced.

Multi-Statement Functions: A function defined over many statements is called a multi-statement function. A multi-statement function begins with the word DEF, the function name, and any parameters, the same as single-statement functions.

Within a multi-statement function definition, an assignment to the function name establishes the value returned when that function evaluation is complete (when FNEND is executed).

If the flow of control through a multi-statement function is such that the function name is not assigned a value, the value returned is the value returned by the previous reference to the function. If the function has not been previously referenced, 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 multi-statement function. See "FNEND Statement" on page 202.

When a multi-statement function is referenced, 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.

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
```

DEFDBL Statement

This statement is provided for compatibility with other BASIC products and is not recommended for new programs. DECIMAL or REAL statements should be used instead. The DEFDBL statement specifies identifiers to be assigned real double or decimal type.

Format

```
DEFDBL [{identifier|(letter-list)} [,{identifier|(letter-list)}]...]
```

where:

identifier
is a 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.

Description

A DEFDBL statement is equivalent to a DECIMAL statement (see "DECIMAL Statement" on page 178) if OPTION ARITHMETIC DECIMAL is in effect and is equivalent to a REAL DOUBLE statement (see "REAL Statement" on page 350) if OPTION ARITHMETIC NATIVE is in effect.

See "OPTION Statement" on page 312 and "SET OPTION Command" on page 483 for information about the ARITHMETIC option.

DEFINT Statement

This statement is provided for compatibility with other BASIC products and is not recommended for new programs. The INTEGER statement should be used instead. The DEFINT statement specifies which identifiers are to be assigned integer type.

Format

```
DEFINT [{identifier|(letter-list)} [,{identifier|(letter-list)}]...]
```

where:

identifier
is a 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.

Description

The syntax and semantics of the DEFINT statement are the same as those for the INTEGER statement (see "INTEGER Statement" on page 258).

DEFSNG Statement

This statement is provided for compatibility with other BASIC products and is not recommended for new programs. DECIMAL or REAL statements should be used instead. The DEFSNG statement specifies identifiers to be assigned real single or decimal type.

Format

```
DEFSNG [{identifier|(letter-list)} [,{identifier|(letter-list)}]...]
```

where:

identifier
is a 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.

Description

A DEFSNG statement is equivalent to a DECIMAL statement (see "DECIMAL Statement" on page 178) if OPTION ARITHMETIC DECIMAL is in effect and is equivalent to a REAL SINGLE statement (see "REAL Statement" on page 350) if OPTION ARITHMETIC NATIVE is in effect.

See "OPTION Statement" on page 312 and "SET OPTION Command" on page 483 for information about the ARITHMETIC option.

DELETE Statement

The DELETE 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 an 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 exception-recovery clauses:

```
EXIT line-reference
IOERR line-reference
NOKEY line-reference
NOREC line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

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.

The file must be opened with access OUTIN and type NATIVE. If the RECORD clause is specified, the file must be opened with organization RELATIVE; if the KEY clause is specified, the file must be opened with organization KEYED. (See "OPEN Statement" on page 305 and "BASIC File Capabilities" on page 61.)

Exception Conditions: Three exception conditions may be recoverable if an exception-recovery 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 exception conditions interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 {numeric-dim | char-dim[*length]} [,{numeric-dim | char-dim[*length]}]...
```

where:

numeric-dim
is a numeric array-declarator.

char-dim
is a character variable name or character array-declarator.

length
is an unsigned, integer constant between 1 and 32767, which defines the maximum string length.

and where:

array-declarator
has the form:

```
array-name (d1[,d2[,d3[,d4[,d5[,d6[,d7]]]]]])
```

where:

array-name
is a numeric or character identifier.

d
Each *d* is an unsigned integer constant in the range 0 to 1 (depending upon the BASE option) to 32767. It defines the upper bound of that dimension.

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 an upper bound in the range 0 to 32767 if the BASE 0 option is in effect, or 1 to 32767 if the BASE 1 option is in effect (see "OPTION Statement" on page 312).

The length of a character variable or array element 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-distributed 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 or in the SUB statement for subprogram units.

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 the default maximum length (assuming the BASE 0 option is in effect).

C$ is a character array having 201 elements, each with a maximum length of 5.

X is a numeric array with 101 elements.

A DIM statement defines the initial numbers of dimensions and the extents of the dimensions for arrays. The number of dimensions and their extents may be changed by a redimensioning clause as long as the initial number of elements is not exceeded. See "Redimensioning" on page 41.

Error Conditions: 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 the BASE 1 option, an error message is printed and the array declaration is ignored.

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.

For the rules regarding the interaction with other immediate statements and program statements, see "Immediate Statements" on page 133 and "Variables and Arrays in Immediate Statements" on page 134.

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 described in "Logical Expressions" on page 57.

Description

The DO statement is used in conjunction with the LOOP statement to define a loop (see "LOOP Statement" on page 268).

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 also "Loop Control Statements" on page 74, and "EXIT IF Statement" on page 200.

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.

Note: The value of the logical expression is checked at the beginning of each loop (it is *not* checked continuously).

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 Block statement is false.

The ELSE block is terminated by the END IF statement.

The ELSE statement is also discussed under "IF Blocks" on page 77, "END IF Statement" on page 194, and "IF Block Statement" on page 234.

END Statement

The END statement indicates both the physical and logical end of the main program.

Format

```
END [numeric-expression]
```

where:

numeric-expression
is 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 system when the program finishes running. The rounded integer value of the expression is returned. If the numeric expression is omitted, zero is returned.

Inside the BASIC environment, the rounded integer value of the numeric expression (if nonzero) 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 informational 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.

The END statement may *not* be followed by any other statements (except for a REM statement) on the same line.

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 77 and "IF Block Statement" on page 234.

See also "ELSE Statement" on page 192.

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 79, "SELECT Statement" on page 366, "CASE Statement" on page 162 and "CASE ELSE Statement" on page 164.

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 93, "SUB Statement" on page 368, "SUBEXIT Statement" on page 370, and "CALL Statement" on page 149.

The END SUB statement may *not* be followed by any other statements (except for a REM statement) on the same line.

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-reference [,condition line-reference] ...
```

where:

condition
is one of the following:

CONV
DUPKEY
DUPREC
ENDPAGE
EOF
IOERR
NOKEY
NOREC
PAGEOFLOW
SOFLOW

No condition may appear more than once.

line-reference
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 exception 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 128. See also "ON Condition Statement" on page 298.

Using an input/output statement with exception-recovery clauses other than EXIT is equivalent to using the statement with an EXIT exception-recovery clause and its corresponding EXIT statement.

An EXIT statement must be the only statement on the line.

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 exception occurs, control is passed to the same locations.

Use of the EXIT statement when a number of statements have the same exception-recovery clauses can reduce the size of your program.

The following list shows conditions for which tests may be made (also refer to Appendix A for complete list of exceptions and conditions):

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 value cannot be converted to the format defined in an associated FORM statement. There are not enough values in the record for the input list items. There is not enough room in the record to write all of the output list items, and SKIP REST is not specified. *Note:* The previous five CONV conditions are for nonstream input/output only.
DUPKEY	A record already exists on the referenced keyed file with the same key as the one specified for the current record.
DUPREC	A record already exists on the referenced relative 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 269 and "ON Condition Statement" on page 298.

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 nonstream file to satisfy an INPUT File, LINE INPUT File, or READ File statement. There is insufficient room in a file to accommodate a PUT, WRITE, PRINT File, or REWRITE statement. A CLOSE statement cannot be completed because of lack of space. A READ statement attempted to read more values than were provided by DATA statements.
IOERR	A hardware malfunction prevents record access and could prevent recognition of other exception conditions. Any input/output exception not covered by one of the other input/output conditions. A format item other than C, V, N, G or PIC in a FORM statement is referred to by a PRINT or PRINT File statement.
NOKEY	No record exists in the referenced keyed file with the key specified.
NOREC	No record exists in the referenced relative 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 127.

For a complete list of exceptions and their corresponding EXIT condition, see Appendix A, "Exception Codes" on page 493.

EXIT IF Statement

The EXIT IF statement is 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 57.

Description

The EXIT IF statement may appear only 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

```
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 through 180 are executed.

Example

```
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
```

The EXIT IF statement at line 190 has no effect provided the logical expression **X = Y** remains **FALSE**. When line 190 is executed and **X** is equal to **Y**, the following actions ensue:

- All statements between the EXIT IF, and the end of the structure enclosing the EXIT IF (the FOR B/NEXT B loop) are bypassed.
- Execution resumes with the first statement after the structure enclosing the EXIT IF, that is, with line 265.

FNEND Statement

The FNEND statement indicates both the physical and logical end of a multi-statement user-defined function.

Format

```
FNEND
```

Description

The FNEND statement marks the physical and logical end of a multi-statement user-defined function. To exit from a multi-statement function, the FNEND statement must be executed.

The FNEND statement must be preceded by a DEF statement.

See "User-Defined Function Statements" on page 90 and "DEF Statement" on page 180.

FOR Statement

The FOR statement initiates a FOR/NEXT count condition loop.

Format

```
FOR var = initial TO final [STEP increment]
```

where:

var
is a simple numeric variable.

initial, final, increment
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 (see "NEXT Statement" on page 297).

The FOR statement *must* be matched with a NEXT statement.

The FOR and NEXT statements are paired, with the same control numeric variable (*var*) 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, *var*, is set to the value of the initial expression, *initial*.

2. If the value of *increment* is positive (or zero) and the value of *initial* is greater than the value of *final*, the loop is not processed, control passes to the first statement following the corresponding NEXT statement, and the value of the control variable remains at the initial value.

   If the value of *increment* is negative and the value of *initial* is less than the value of *final*, the loop is not processed, control passes to the first statement following the corresponding NEXT statement, and the value of the control variable remains at the initial value.

3. If the value of *increment* is positive (or zero) and the value of *initial* is less than or equal to the value of *final*, the statements in the loop are processed, and the value of *increment* is added to the control variable when the NEXT statement is executed.

   If the value of *increment* is negative and the value of *initial* is greater than or equal to the value of *final*, the statements in the loop are processed, and the value of *increment* is added to the control variable when the NEXT statement is executed.

   If STEP *increment* is omitted, a step increment of 1 is used.

4. The body of the FOR loop is repeated and continues until the control variable is greater than (or, for a negative increment, less than) the value of *final*. The control variable remains at this value.

5. Control now passes to the first statement following the corresponding NEXT statement.

The control variable may be altered within the FOR/NEXT loop (for example, by appearing on the left of an equal sign of an assignment statement), but this is not recommended. The same control variable may not be used as the control variable for another FOR/NEXT loop nested within the FOR/NEXT loop of the control variable.

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, END SUB, RETRY, RETURN, and SUBEXIT 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 74 and "EXIT IF Statement" on page 200.

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 14 times. During the first iteration, FEET is 9, during the second iteration, FEET is 12, and so forth. 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 a

```
literal
control-specification
[repeat*]data-form
```

And where:

literal
is a quoted character string.

control-specification
is one of the following:

`X[e]` skips *e* positions in record or on line.

`POS[e]` positions to location *e* in record or on line.

`SKIP[e]` skips *e* number of lines.

`[NEW]PAGE` positions to the top of a new page.

where:

e
is a numeric expression, the value of which is rounded to the nearest integer. It may not refer to user-defined functions.

repeat
is an optional unsigned, nonzero integer constant or a simple numeric variable, whose value is used as a replication factor with a data-form. Its rounded integer value must be greater than zero.

data-form
is one of the data-forms shown in Figure 32 on page 206.

Data-form	Meaning
PIC (s[s]...[¬¬¬[¬...]][tr])	Picture of data item
C[w]	Character data
V[w]	Character data with trailing spaces removed on input
Nw[.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
NCw[.d]	Conversion of numeric data to zoned decimal format on output, and conversion of either zoned decimal or numeric characters on input
ZDw[.d]	Conversion of zoned decimal data for both input and output
B[w]	Fixed-point binary
S	Short-form floating-point binary (32 bits) - real single
L	Long-form floating-point binary (64 bits) - real double
PDw[.d]	Packed decimal
ND	Internal decimal
NI	Internal integer

where:

w
is an unsigned, nonzero integer constant, which may be preceded by spaces.

d
is an unsigned, integer constant.

s
is a digit specifier (#, Z, *, $, +, or -), or an insertion character. An insertion character is a comma (,), slash (/), B, or decimal point (.).

¬
is an exponent specifier. (This is the circumflex character on some terminals and printers.)

tr
is a trailing character. A character is a trailing plus (+), trailing minus (-), trailing credit (CR), or either form of trailing debit (DB or DR).

Figure 32. FORM Statement Data-Form Codes

Description

The FORM statement is used in conjunction with the PRINT, PRINT File, READ File, REREAD, WRITE, and REWRITE statements. These statements may reference the line number or line label of a FORM statement, or specify a character expression which evaluates to a FORM statement.

The data-forms C, V, N and G may also be used in PRINT FIELDS and INPUT FIELDS statements.

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.

A FORM must be the only statement on a line.

Literal Specifications

If a quoted character string appears in a FORM associated with a READ File or REREAD statement, it is ignored if it is a null string. Otherwise, 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 ignored if it is a null string. Otherwise, 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 quotation marks). 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 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 expression. When these control specifications are used in a FORM statement, the expression may be any numeric expression. However, when an input/output statement uses a character expression as a FORM (the USING clause specifies a character expression rather than a FORM statement), a numeric expression for a control specification within the character string must be an unsigned integer constant.

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.

e is a numeric expression, the value of which is rounded to the nearest integer. It may not refer to user-defined functions.

e may be preceded by spaces. At least one space must precede e if e begins with a variable name or if e begins with a constant followed by an asterisk.

For example, in a FORM statement,

```
X100*NI
```

would be interpreted as the data-form specification NI repeated for the value of the variable X100. If the desired X control specification is 100 times the variable NI, the FORM should be written as follows:

```
X 100*NI
```

If the rounded value of e is less than one, no positions are skipped. If e is omitted, one position is skipped.

For READ File, REREAD, WRITE, and REWRITE, if the resulting position is greater than the record length, a CONV exception occurs.

For PRINT and PRINT File operations, if the resulting 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 at 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, the value of which is rounded to the nearest integer. It may not refer to a user-defined function.

e may be preceded by spaces. At least one space must precede e if e begins with a variable name or if e begins with a constant followed by an asterisk. For example, in a FORM statement, the specification:

```
POS100*NI
```

would be interpreted as the data-form specification NI repeated for the value of the variable POS100, not as the POS control specification with the expression 100 times the variable NI, which should be written as:

```
POS 100*NI
```

For READ File, REREAD, WRITE, and REWRITE operations, if the rounded value of e is less than 1 or is omitted, 1 is used; if the rounded value of e is greater than the record length, a CONV error occurs.

For PRINT and PRINT File operations, if e is omitted or the rounded value of e is less than 1, the left margin value is used. If the rounded value of e is greater than zero but less than the left margin, an exception occurs (the SYSTEM action is to print a message and continue execution using the left margin value). If the rounded value of 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 (see "MARGIN Statement" on page 269).

Example

```
100 WRITE #10 USING 200 : A$,B%
200 FORM C15,POS 24,N6
```

The value of A$ will be written in positions 1 through 15; B% will be written in positions 24 through 29.

The position used may be less than the previous position in the record, in which case subsequent values output to the record will replace existing values, or previous values that have been read will be reused.

Example

```
 * list
100 rem print graph of cosine and sine functions between
110 rem -pi and pi
120 def func1(x)=cos(x)
130 def func2(x)=sin(x)
140 def scale(y)=ifix(y*10)+22
150 print using 160
160 form pos 11,'* = cosine, @=sine'
170 print
180 print using 190
190 form pos 11,'-1',pos 22,'0',pos 32,'1'
200 for x=-pi to +pi step pi/8
210   if round(x,2)=0 then a$='-' else a$=' '
220   y1 = scale(func1(x)) : y2 = scale(func2(x))
230   print using 240: rpt$(a$,23),x,'*','@'
240   form pos(11),c 23,pos 1,n 8.2,pos(22),'|',pos(y1),c1,pos(y2),c1
250 next x
260 end
 * run
         * = cosine, @=sine

         -1          0          1
   -3.14  *          @
   -2.75    *    @   |
   -2.36       @     |
   -1.96     @   *   |
   -1.57    @        *
   -1.18     @       |   *
    -.79       @     |       *
    -.39          @  |         *
     .00  -----------@---------*-
     .39             |   @     *
     .79             |       @
    1.18             |   *     @
    1.57             *           @
    1.96          *  |           @
    2.36        *    |       @
    2.75      *      |   @
    3.14    *        @
END AT LINE 260.
```

This example demonstrates the use of POS in a FORM statement in order to print a graph of two functions. Notice that POS may be used to position to a previous point in a record.

SKIP The SKIP[e] control specification indicates how many lines are to be skipped before the next value is printed; this control specification is valid only with the PRINT or PRINT File statement.

e is a numeric expression, the value of which is rounded to the nearest integer. It may not refer to user-defined functions.

e may be preceded by spaces. If e begins with a variable name or a numeric constant followed by an asterisk, at least one space must separate SKIP and e. For example, in a FORM statement, the specification:

```
SKIP100*NI
```

would be interpreted as the data-form specification NI repeated for the value SKIP100. If what is desired is the SKIP control specification with the expression 100 times the variable NI, the specification should be written as:

```
SKIP 100*NI
```

If the rounded value of e is less than 0 or is omitted, the value 1 is used.

If the rounded value of 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 (the file is opened as DEVICE PRINTER or DEVICE 3800, see "OPEN Statement" on page 305), 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. This is useful for underlining.

- If the file does not have carriage control characters or if the option is to PRINT to the terminal, SKIP 0 is the same as POS (with e omitted); characters are overlaid in the current record but the effect is replacement rather than overprinting.

When the rounded value of 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 the rounded value of 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 269 and "ON Condition Statement" on page 298).

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. NEWPAGE may be used instead of PAGE and has the same effect.

If PAGE is specified when output is to a file, and the file is not opened as DEVICE PRINTER or DEVICE 3800 (see "OPEN Statement" on page 305), an exception is generated. The SYSTEM action for the exception is a warning message.

When used with a PRINT or PRINT File 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 269.

Example

```
100 PRINT #5 USING 200 : A$,B%
200 FORM C10,PAGE,N5
```

The value of B% will appear at the top of the page following the page on which A$ appears.

Note: A PAGE control specification will *not* cause an ENDPAGE condition to be generated (see "ON Condition Statement" on page 298).

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. Within a FORM statement, the repeat count can be either an unsigned, nonzero integer constant or a simple numeric variable. But, within FORM specifications that are character strings specified in the USING clause of input/output statements, the repeat counts must be integer constants only.

If the repeat count is a simple numeric variable, its rounded integer value is used as the repeat count. The rounded value must be greater than zero.

A data-form with a repeat count is equivalent to that data-form occurring the repeat count number of times in a row.

The following figure shows which data-form specifications may be used in FORM statements referenced by each I/O statement.

	PIC	C	V	N	G	NC	ZD	B	S	L	PD	ND	NI
PRINT	X	X	X	X	X								
PRINT File	X	X	X	X	X								
WRITE	X	X	X	X	X	X	X	X	X	X	X	X	X
REWRITE	X	X	X	X	X	X	X	X	X	X	X	X	X
READ File		X	X	X	X	X	X	X	X	X	X	X	X
REREAD		X	X	X	X	X	X	X	X	X	X	X	X
PRINT FIELDS	X	X	X	X	X								
INPUT FIELDS		X	X	X	X								

Figure 33. I/O Statements and their Respective PIC Data-Form Specifications

PIC

The PIC data-form specification may only be used on FORM statements associated with the output statements PRINT, PRINT File, WRITE, and REWRITE and in the PRINT FIELDS statement.

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.

The picture field may not have leading or trailing spaces (the first character of the picture field must be immediately preceded by a left parenthesis and the last character must be immediately followed by a right parenthesis). A picture field may not contain spaces. A picture field must contain at least one digit specifier (#, Z, *, $, +, -) and for $, +, and -, at least one other digit specifier must also occur.

The picture syntax is not checked until the FORM is used. It is not checked by the compiler (the compiler allows any characters between the parentheses).

For character data, the contents of the picture field are ignored (an exception occurs if the picture field does not have valid syntax) 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 spaces on the right. If the character value is longer than the picture field, a string overflow (SOFLOW) exception occurs.

For numeric data, the picture field specifies both the 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 ("␣" in output examples implies a space):

Specifies a data position where a digit must always appear.

```
                                Value       Output
100 FORM PIC (#####)             63         00063
```

Z A leading zero in the associated data position is replaced by a space.

If the value is 0 and the picture field does not contain a #, all the positions are filled with spaces.

```
                                Value       Output
110 FORM PIC (ZZZ.##)           24.6        ␣24.60
                                 0          ␣␣␣00

115 FORM PIC (ZZZ.ZZ)           24.6        ␣24.60
                                 0          ␣␣␣␣␣␣
```

If the digit specifier preceeding a decimal point is a Z it will always have a digit (0 if there is no significant digit to the left of the decimal point), unless the entire field is zero. In that case, the zero to the left of the decimal point is omitted.

A Z may not follow any # digit specifiers in a picture field.

***** A leading zero in the associated position is replaced by an asterisk. If the value is 0 and the picture field does not contain a #, all positions are filled with asterisks.

```
                                Value       Output
120 FORM PIC (***##)             243        **243
                                  -6        -**06
                                   0        ***00

125 FORM PIC (*****)             243        **243
                                  -6        -***6
                                   0        *****

130 FORM PIC (***.##)              0        ***.00
                                0.01        **0.01

135 FORM PIC (***.**)              0        ******
                                0.01        **0.01
```

If the digit specifier preceding a decimal point is an * it will always have a digit (0 if there is no significant digit to the left of the decimal point), unless the entire field is zero. In that case, the zero to the left of the decimal point is omitted and the entire field is replaced with asterisks.

Note: The use of both Z and * in a picture field is not valid.

An * may *not* follow any # digit specifier in a picture field.

$ Specifies each data position that can potentially be occupied by a floating currency sign (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 $, comma, or slash appeared in the digit specifier and where no significant digit is present to the left.

If the value is 0 and the picture field does not contain a #, all the positions are filled with asterisks if an * appears in a picture field; otherwise it is filled with spaces.

```
                                Value       Output
130 FORM PIC ($$$##)             243        ø$243
                                  -6        ø$-06
```

A $ may *not* follow a Z, *, or # in a picture field.

Note: Use of a $ digit specifier has the effect of suppressing nonsignificant leading zeros in the associated position.

+ Specifies each data position where a floating leading 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        Ø+243
                                -6        ØØ-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, comma, or slash appeared in the digit specifier, and where no significant digit is present to the left.

Only a single leading + may precede a $ in the picture field. Only a single $ may precede a leading + in a picture field. A leading + may *not* follow a Z, *, or # in a picture field.

If the value is 0 and the picture field does not contain a #, all the positions are filled with asterisks if an * appears in a picture field; otherwise it is filled with spaces.

Note: Use of a + digit specifier has the effect of suppressing nonsignificant leading zeros in the associated position.

- Specifies each data position where a floating leading minus sign may appear if the field is negative.

```
                              Value       Output
150 FORM PIC (---##)           243        ØØ243
                                -6        ØØ-06
```

If a single - is used in the picture field, 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 picture field, the sign of the numeric value appears, justified as far to the right as possible, wherever a minus sign, comma, or slash appeared in the digit specifier, and where no significant digit is present to the left.

Only a single leading - may precede a $ in a picture field. Only a single $ may precede a leading - in a picture field. A leading - may *not* follow a Z, *, or # in a picture field. A leading - is not allowed with a leading + in a picture field.

If the value is 0 and the picture field does not contain a #, all the positions are filled with asterisks if an * appears in a picture field; otherwise it is filled with spaces.

Note: Use of a - digit specifier has the effect of suppressing nonsignificant leading zeros in the associated position.

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 to

the left of the rightmost digit (or instead of the leftmost zero for # digit specifiers). A floating $ will be moved to the left one position if it would normally occur in this position.

Insertion characters are characters inserted into the output at the position they are specified in the picture field. Insertion characters may be either conditional or unconditional.

The unconditional insertion character is:

B Space, which always appears when specified.

The conditional insertion characters are:

, Comma

/ Slash

. Decimal point

The rules for the appearance of the insertion characters are:

- A comma or a slash is not allowed as the first character in the picture field. (The condition for their appearance can never be met.)
- A comma or a slash is not allowed after a decimal point.
- If a # appears in the picture field, or if a significant digit is found to their left, the comma and slash always appear.
- If a # does *not* appear in the picture field then, if no significant digit exists to their left, a comma or slash do not appear. They are replaced, instead, by the character that would be used for the preceding digit specifier.

  The decimal point (.) always appears in its associated position except in the following circumstance: the picture field does *not* contain a # and the value of the numeric field is zero. In this case, the decimal point is replaced by an * if the picture field contains a # and a space, if otherwise.

The trailing characters are:

CR Trailing credit symbol

DB or DR Trailing debit symbol

+ or - Trailing sign symbols

The trailing symbols (CR, DB, DR, trailing +, trailing -) are replaced by either spaces or asterisks if the conditions for their appearance are not met (two spaces or two asterisks for CR, DB, and DR, otherwise 1 space or 1 asterisk). Only one trailing symbol representation is

permitted per picture field, and then only if there is no leading + or -, and there is no exponent specifier.

- The trailing plus sign always causes the appearance of either a positive or a negative sign, unless the value is zero and the picture field does not contain a # but does contain an *, in which case an asterisk appears.

- The other trailing symbols (minus sign, CR symbol, DB or DR symbol), appear only when the resulting value is negative. If the value is zero, and the picture field does not contain a # but does contain an *, the trailing symbol is replaced by *.

Three or more occurrences of the exponent specifier (¬) specify the presence in the corresponding print positions of the following sequence:

1. The letter E

2. The sign of the exponent (+ 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 previously specified will therefore always appear in a field defined with an exponent specifier.

PIC Data-Form Examples

The following examples show some of the different types of conversion specifications that can be used with the PIC data-form. Spaces in the result field are shown here as the lowercase "ƀ."

- Integer Format:

```
Value       Specification          Result

   10       PIC(####)              0010
   10       PIC(ZZZZ)              ƀƀ10
   10       PIC(****)              **10
   10       PIC($$$$)              ƀ$10
  -10       PIC($$$$$$$$)          ƀƀƀƀ$-10
20289       PIC(##/##/##)          02/02/89
 -123       PIC(ZZZZDB)            ƀ123DB
  123       PIC(++++ZZZZ)          ƀƀƀ+ƀ123
  123       PIC(+ZZZZZZZ)          +ƀƀƀƀ123
```

- Fixed-Point Format:

```
Value       Specification          Result

12.145      PIC(ZZZ.##)            ƀ12.15
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 exception (SOFLOW) occurs. If n is greater than w, the variable's length becomes the length of the transmitted character string. 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 spaces are added to the right of the expression's value. If n is greater than w, a string overflow (SOFLOW) occurs. The value of w defaults to 1.

This data-form specification is valid for INPUT FIELDS, PRINT, PRINT FIELDS, PRINT File, READ File, REREAD, REWRITE, and WRITE statements.

See examples on page 219.

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 (trailing spaces are removed). If the variable's maximum length, n, is less than the number of characters moved, p, a string overflow exception 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.

This data-form specification is valid for INPUT FIELDS, PRINT, PRINT FIELDS, PRINT File, READ File, REREAD, REWRITE, and WRITE statements.

See examples on page 219.

Nw[.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 (with rounding) to 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 spaces, the number is set to 0. Spaces are allowed in the field after the first numeric character, but not imbedded in the value. 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). The position is determined from the rightmost nonblank character in the field.

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.

If the value of the numeric expression is positive, the value of d must be less than or equal to w-1. If the value of the numeric expression is negative, the value of d must be less than or equal to w-2. 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 INPUT FIELDS, PRINT, PRINT FIELDS, PRINT File, READ File, REREAD, REWRITE, and WRITE statements.

See examples on page 219.

G[w[.d]] This data-form specification can be used to transmit either numeric or character values. 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 and 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 C specification, and 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 INPUT FIELDS, PRINT, PRINT FIELDS, PRINT File, READ File, REREAD, REWRITE, and WRITE statements.

Data Conversion Examples

The following are examples of the C, V, N, and G conversion specifications. The character "␣" shows spaces in the result.

Output:

Value	Specification	Result (Characters)
AB	C3	AB␣
AB	V4	AB␣␣
AB	C	string overflow exception
AB	G7.3	AB␣␣␣␣␣
75	N3	␣75
3.45	N7.2	␣␣␣3.45
3.45	N7.1	␣␣␣␣3.5
-3.45	N7	␣␣␣␣␣-3
3.45	G7.3	␣␣3.450

Input:

Value	Specification	Result (Characters)
AB␣	C3	AB␣
AB␣␣	V4	AB
AB	C	A
AB	G7.3	AB␣␣␣␣␣
␣75	N3	75
␣␣␣3.45	N7.2	3.45
␣␣␣␣␣-3	N7	-3
3.45	G7.3	␣␣3.450

NCw[.d] On input, indicates that the next w positions of a record contain a numeric value in zoned decimal format, or the value that was written by a PIC data-form specification. The value is converted (with rounding as needed) to the type of the receiving numeric 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
space
comma (,)
decimal point (.)
exponential notation (E plus or minus numeric constant)

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 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 File, REREAD, WRITE, and REWRITE statements. See examples under ZD on page 221.

ZDw[.d] On input, the next w positions of the record contain a numeric value in zoned decimal form (a zone and a digit for each position, except for the low-order position, which may contain either a zone or a sign and a digit). This numeric value is to be converted to internal numeric representation and moved to a corresponding numeric variable in the input list.

The optional d specifies the number of digits in the fractional portion of the number, and is assumed to be zero if absent.

On output, the ZD specification acts exactly as the NC specification.

If the optional 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 File, 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
␣␣$1,234.56CR	NC13.3	-123.456
␣␣$1,234.56CR	ZD13.3	exception
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 C, "Character Set Collating Sequences" on page 505 for the characters these codes represent.

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 File, REREAD, WRITE, and REWRITE statements.

S On input, S 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, real double, or integer format as required by the receiving variable.

On output, S 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 File, REREAD, WRITE, and REWRITE statements.

L On input, L 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, real single, or integer format is performed as required by the receiving variable.

On output, L specifies that a numeric value is to be converted to the long form of a floating-point operand and written to the record.

This data-form specification is valid for READ File, REREAD, WRITE, and REWRITE statements.

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.

PDw[.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 (the packed decimal value is adjusted by dividing it by 10**d).

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 is right-justified in the packed decimal field with the designated number of fractional digits with any excess digits truncated. If any nonzero digits of the integer part or remaining fractional part would extend past the left end of the field an exception occurs. (The numeric value is multiplied by 10**d and the integer part is output as a packed decimal number.)

The value of d must be less than or equal to the value of 4*w.

This data-form specification is valid for READ File, REREAD, WRITE, and REWRITE statements.

Packed Decimal Conversion Example

Value	Specification	Result (Hexadecimal)
3.45	PD 7.2	0000000000345C
3.37	PD 7.1	0000000000034C
-3.29	PD 7	0000000000003D
3.81	PD7.12	3810000000000C
.00267	PD7.14	0267000000000C
.00267	PD7.16	exception

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 or real type, the value is converted to integer or real format with rounding.

On output, a value is written to the next 12 character positions in the record in internal floating decimal format. Integer or real values are converted to their decimal equivalent.

This data-form specification is valid for READ File, 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 or real type, the value is converted to decimal or real format.

On output, a numeric value is written to the record in the next 4 character positions in internal integer format. Decimal or real values are rounded and converted to their integer equivalent.

This data specification is equivalent to B4 (on page 222).

This data-form specification is valid for READ File, REREAD, WRITE, and REWRITE statements.

Internal Integer Conversion Example

```
050 OPTION ARITHMETIC DECIMAL
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 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 sequential internal file.

Format

```
[MAT] GET #fileref : input-list [,SKIP REST] [err [,err]...]
```

where:

fileref
is a numeric expression which, when evaluated and rounded to an integer, is within the range 1 to 255. It identifies the file to be processed.

input-list
is a list of one or more variables, array elements, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
EOF line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

Description

For stream files, (opened with STREAM or SEQUENTIAL organization), 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 sequential files (opened with INTERNAL type and SEQUENTIAL organization), the GET statement retrieves one record and assigns each of its values to the corresponding items of the input list (if necessary, additional records are read to satisfy the input list).

An internal sequential 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 (each value in a record is treated as if it occurred separately, except that any values not used from a record are ignored; that is, another GET will start from the next record).

Each value retrieved and assigned must be of the same 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, real, 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.

For more information see "BASIC File Capabilities" on page 61.

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 name in the input list must be preceded by the MAT keyword. See "Input/Output Lists" on page 85 for more information.

Fileref: The file reference must refer to an INTERNAL type file (with either STREAM or SEQUENTIAL organization). See "Combinations of File Type, Organization, Access, and Records" on page 64, and "OPEN Statement" on page 305.

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 elements may not be exceeded. For more information, see "Input/Output Data Rules" on page 86.

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 exception occurs.

If there are no more values in the file to assign to remaining input list items, an end-of-file (EOF) condition exists.

SKIP REST Clause: For an internal sequential file opened with SEQUENTIAL organization, 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.

For an internal sequential file opened with STREAM organization, the SKIP REST clause has no meaning and is ignored. Excess values in a record are ignored whether or not a SKIP REST clause is specified.

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.

Exception Conditions: The string overflow (SOFLOW), conversion (CONV), and end-of-file (EOF) conditions described above, as well as the input/output exception condition (IOERR), may be recoverable if the corresponding exception-recovery 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 exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128. See also "EXIT Statement" on page 197.

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 assigns values to the newly-dimensioned array.

GOSUB Statement

The GOSUB statement is used, together with the RETURN statement, to invoke subroutines.

Format

```
GOSUB line-reference
```

where:

line-reference
is a line number or line label.

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 statement following the last GOSUB statement which was executed (see "RETURN Statement" on page 361).

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 after 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 298 and "Exception Handling Statements" on page 127.

Processing of a RETURN statement without an active GOSUB statement results in an exception.

More than one GOSUB statement may be active; a 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 multi-statement function are set inactive by the execution of a SUBEXIT, an END SUB, or an FNEND statement.

See also "Subroutine Control Statements" on page 73 and "ON GOTO/GOSUB Statement" on page 303.

The following example shows the sequence of execution of a program that has GOSUB and RETURN statements. Note that GOSUB statements may appear in subroutines as well as in the main program. Therefore, subroutines SUB_A and SUB_B can call subroutine SUB_C, just as the main program calls each of these subroutines. The RETURN statements in subroutines SUB_A, SUB_B, and SUB_C cause execution to continue at the line after the last GOSUB executed.

```
0240 !********************* MAIN *********************************
0260   PRINT 'MAIN INVOKES SUB_A'
0270   GOSUB SUB_A
0280   PRINT 'MAIN RESUMES AFTER RETURN FROM SUB_A'
0290   PRINT
0300   PRINT 'MAIN INVOKES SUB_B'
0310   GOSUB SUB_B
0320   PRINT 'MAIN RESUMES AFTER RETURN FROM SUB_B'
0330   PRINT
0340   PRINT 'MAIN INVOKES SUB_C'
0350   GOSUB SUB_C
0360   PRINT 'MAIN RESUMES AFTER RETURN FROM SUB_C'
0910   GOTO 9999
1000 !********************* SUB_A ********************************
1004   SUB_A:                                  ! GOSUB DESTINATION
1020   PRINT TAB(5); 'SUB_A INVOKES SUB_C'
1030   GOSUB SUB_C
1040   PRINT TAB(5); 'SUB_A RESUMES AFTER RETURN FROM SUB_C'
1060   RETURN
2000 !********************* SUB_B ********************************
2004   SUB_B:                                  ! GOSUB DESTINATION
2020   PRINT TAB(5); 'SUB_B INVOKES SUB_C'
2030   GOSUB SUB_C
2040   PRINT TAB(5); 'SUB_B RESUMES AFTER RETURN FROM SUB_C'
2060   RETURN
3000 !********************* SUB_C ********************************
3004   SUB_C:                                  ! GOSUB DESTINATION
3020   PRINT TAB(10); 'SUB_C EXECUTES'
3040   RETURN
3070 !*************************************************************
9999   END
```

When this program is executed, the output will be as follows:

```
MAIN INVOKES SUB_A
    SUB_A INVOKES SUB_C
         SUB_C EXECUTES
    SUB_A RESUMES AFTER RETURN FROM SUB_C
MAIN RESUMES AFTER RETURN FROM SUB_A

MAIN INVOKES SUB_B
    SUB_B INVOKES SUB_C
         SUB_C EXECUTES
    SUB_B RESUMES AFTER RETURN FROM SUB_C
MAIN RESUMES AFTER RETURN FROM SUB_B

MAIN INVOKES SUB_C
         SUB_C EXECUTES
MAIN RESUMES AFTER RETURN FROM SUB_C
```

GOTO Statement

The GOTO statement unconditionally branches to the indicated line number or line label.

Format

```
GOTO line-reference
```

where:

line-reference
is a line number or line label.

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 298 and "Exception Handling Statements" on page 127.

Example

```
100 GOTO PART_2
.
.
.
190 PART_2: LET A = B+C
```

When statement 100 is executed, all statements between 100 and 190 (PART__2) are bypassed. Sequential processing continues from that point. See also "Control Statements" on page 73 and "ON GOTO/GOSUB Statement" on page 303.

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 57.

statement-reference
is a line number, line label, or list of imperative statements separated by colons.

Description

The logical expression in the IF statement is evaluated. When the expression is true:

- if a line number or label follows the THEN, execution branches to the specified line.
- if imperative statements follow the THEN as part of the IF statement, the statements are executed, and control passes to the next line (the ELSE clause is skipped if present).

When the logical expression is false, and an ELSE clause is present:

- if a line number or label follows the ELSE, execution branches to the specified line.
- if imperative statements follow the ELSE as part of the IF statement, the statements are executed, and control passes to the next line.

When the logical expression is false and an ELSE clause is not present, execution continues at the next line after the IF statement.

Example

```
200 IF A = B THEN EQUAL
210 X = X+Y
```

If the condition is true (the values of A and B are equal), execution branches to the line labeled EQUAL. Otherwise, execution continues at line 210.

Example

```
100 IF A=B THEN LET C=D: LET E=F: LET G=H
```

If the condition is true (the values of A and B are equal), the three LET statements in the list of statements are executed. Otherwise, no action is taken by the IF statement and execution continues at the next line.

Example

```
200 IF A = B THEN EQUAL ELSE LET C = E
210 X = X+Y
```

If the condition is true (the values of A and B are equal), execution branches to the line labeled EQUAL and continues from that point. Otherwise, the LET statement is executed and execution continues at line 210.

When an IF statement ends with a statement list, all of the remaining 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 NEXT
210 X = X + Y
```

is not equivalent to

```
200 IF A = B THEN GOTO NEXT: X = X + Y
```

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 34 on page 233.

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 34. Imperative Statements

If you wish to execute statements other than imperative statements, use the "IF Block Statement" on page 234.

IF Block Statement

The IF Block statement begins an IF block.

The IF Block statement evaluates a logical expression and conditionally transfers control, based on whether the logical expression is true or false.

Format

```
IF logical-expression THEN
```

where:

logical-expression
can be any logical expression as described in "Logical Expressions" on page 57.

Description

The IF Block statement is used with the END IF statement to construct IF blocks, which may be subdivided into THEN and ELSE blocks. The interrelationship of these statements is discussed under "Decision Structure Control Statements" on page 77 and "IF Blocks" on page 77. See also "ELSE Statement" on page 192 and "END IF Statement" on page 194.

The IF block consists of five basic parts:

1. The IF Block statement containing the logical expression to be evaluated and the keyword THEN.
2. The THEN block, which includes all the statements between the IF Block statement and either the optional ELSE statement or the END IF statement.
3. The optional ELSE statement, which contains only the keyword ELSE.
4. The optional ELSE block, which includes all statements between the ELSE statement and the END IF statement.
5. The END IF statement, which terminates the IF block.

The major differences between an IF statement (see "IF Statement" on page 231) and an IF Block statement are:

- An IF Block statement requires a matching END IF statement. The END IF statement is invalid for an IF statement.
- In an IF Block statement, the statements following THEN or ELSE may be coded on separate lines. In an IF statement, all the statements following both THEN and ELSE must be imperative statements and must be contained on a single line (or continuation line).

IF Block Statement Example

```
200  IF A = B THEN
210        C = D
220        E = F
230        G = H
240     ELSE
250        C = A
260        E = B
270  END IF
280  X = X+Y
```

IF Statement Example

```
200  IF A = B THEN &
  &        C = D : &
  &        E = F : &
  &        G = H   &
  &     ELSE       &
  &        C = A : &
  &        E = B
210 X = X+Y
```

These two code sequences are functionally equivalent:

compact.

- If A equals B, C is set equal to D, E to F, and G to H.
- If A does not equal B, C is set equal to A, and E to B.

Note: After the IF Block or IF statement is executed, X=X+Y is executed.

One of the major advantages of block structuring is that structures may be nested within other structures. The following example illustrates this capability of the IF Block statement. The IF Block statement at line 190 begins a logical selective structure which continues until line 320. The THEN block (lines 220 through 260) contains another IF Block. The ELSE block (lines 290 through 310) contains a FOR - NEXT repetition.

Nesting Within an IF Block Example

```
0190   IF logical expression THEN
0220         IF logical expression THEN
                    .
                    .
                    .
0240              ELSE
                    .
                    .
                    .
0260         END IF
0270      ELSE
0290         FOR X = 1 TO 10
                 .
                 .
                 .
0310         NEXT X
0320   END IF
```

IMAGE Statement

The IMAGE statement specifies the format of print lines or output records for display files.

Format

```
[IMAGE]:[character-string][conversion-specification [character-string]]...
```

where:

character-string
is a string of characters, which may contain quotes but may not contain characters that may be interpreted as the start of a conversion-specification.

conversion-specification
is of the form:

```
[justifier] [floating-header]
     {i-format-item | f-format-item | e-format-item}
```

where:

justifier
can be the character < or >

floating-header
can be the characters

```
+[+...], -[-...], or $[$...]
```

i-format-item
representing integer notation, is in the form:

```
[digit-spec] [digit-spec] digit-spec
[[,]digit-spec digit-spec digit-spec]...
```

and where

digit-spec
can be

```
#, %, or *
```

In an i-format-item, all digit specifiers (#,%,*) must be the same character.

f-format-item
representing fixed-point notation, is in the form:

```
{.#[#]... | i-format-item.[#]...}
```

e-format-item

representing floating-point notation, is in the form:

```
{i-format-item | f-format-item}¬¬¬[¬]...
```

The circumflex character (^) and the logical NOT (¬) are equivalent. Use the one your terminal can generate and display.

Spaces between the optional keyword IMAGE and the colon are ignored.

Spaces following the colon and preceding nonblank data in the image represent space characters, and are significant.

Spaces following the last nonblank character in the image are not significant, and are ignored.

A conversion-specification may not contain spaces.

Description

The IMAGE statement, in conjunction with PRINT USING statements, specifies the formatting of all or part of a print line. In conjunction with PRINT USING File statements, the IMAGE statement specifies the formatting of output records for display files.

IMAGE statements are not executable and may appear anywhere in a program unit. An IMAGE statement may be continued, but the continuation ampersands, any spaces between the ampersands, and end-of-record characters are not considered part of the image.

An IMAGE statement must be the only statement on a line.

The items specified in the output list of the PRINT USING or PRINT USING File statement interact with the character strings and conversion specifications of the image to create the output line or record. Character strings appearing in the image are displayed exactly as they appear. Each scalar reference of the output list is edited in order of appearance into the corresponding conversion specification. Conversion specifications indicate Integer, Fixed-Point, or Floating-Point notation.

Character Conversion

A character item in the output list may be written with any conversion specification. The character string value is placed in the line, replacing every element in the conversion specification (including +,-, comma, <, >, #, ., %, *, $, and ¬).

If the character string is shorter than the conversion specification, padding with spaces occurs. If a < character is present, indicating left-justification of the field, padding occurs on the right side; if a > character is present, representing right-justification, padding occurs on the left side. If neither the < or > character is present, the character string is centered and padding occurs on both sides; if the character string cannot be centered exactly, the right side will contain one more space than the left side.

If a character string contains no characters (is null), the entire conversion specification is filled with spaces. A string overflow (SOFLOW) exception occurs if a string value is longer than the conversion specification.

In the following example, the character ␣ represents a space.

Character String Conversion Example

```
Value              Specification       Result

ABC                <#####              ABC␣␣␣
ABC                >$$$##              ␣␣␣ABC
ABC                ######              ␣ABC␣␣
WRONG              ###.## ¬¬¬          ␣␣WRONG␣␣
```

Numeric Conversion

A numeric item in the output list is converted to output format in accordance with the conversion specification as follows:

- All numeric values are placed in the line right-justified; therefore, a justifier (< or >) in the conversion specification has no special meaning. If a justifier is present, it is replaced by the floating character (+, -, or $) if present, or else by a digit specifier of the same type as the first digit specifier on its right.

  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 fixed-point notation, rounding the value or extending it with zeros in accordance with the conversion specifications.

  E-format

  The value is converted to floating-point notation, 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 floating-point notation.

  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 35. IMAGE Statement Format Specification

If the exponent exceeds the width provided, an exception occurs.

- The converted numeric value is edited according 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 36 on page 240, "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 below.

Floating-Header	Sign of Value	Floating Symbol
+[+]...	positive	+
+[+]...	negative	-
-[-]...	positive	space
-[-]...	negative	-
$[$]...	positive	$
$[$]...	negative	$-

Figure 36. 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            ####               ß-12
-12            ****               -*12
-12            %%%%               -012
```

  (The "ß" 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 spaces.

  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 filled with asterisks.

When the floating-header is $, the position where the minus sign is displayed 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	$$$$####	␣␣␣$-123
-12	$$$####	␣␣$␣-12
-12	$$$****	␣$-**12
-12	$$$%%%%	␣$-0012
-12	$****	$-*12
-12	$%%%%	$-012
-12	$####	$␣-12

(The "␣" shows a blank (space) position in the result.)

Format Conversion Examples

The following are examples of the different types of conversion specifications. The letter "␣" shows blank (space) in the result.

I-format Examples:

Value	Specification	Result
1000000	##,###,###	␣1,000,000
-99999	##,###,###	␣␣␣-99,999
3	****	***3
3.2	****	***3
100	***,***	****100
4	%%%%	0004

F-format Examples:

Value	Specification	Result
12.145	###.##	␣12.15
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 "INPUT FIELDS Statement" on page 248 and "INPUT File Statement" on page 255).

Format

```
[MAT] INPUT [input-prompt:] input-list [err [,err]...]
```

where:

input-prompt
is one of the following:

```
PROMPT character-expression
PROMPT quoted-character-constant
quoted-character-constant
```

input-list
is a list of one or more variables, array elements, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

Description

When at a terminal, this form of the INPUT statement allows the user to provide values for program variables from the terminal during program execution.

In batch mode, this form of the INPUT statement is system dependent. See your IBM BASIC 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 85 for more information.

Input-Prompt: If an input prompt is present, the value of the character 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 that 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 of 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 exception-recovery clause, control is transferred to the statement specified in the clause. If there is no corresponding exception-recovery clause, a request is made that the current input reply be reentered.

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 if the INPUT statement contains an appropriate exception-recovery clause, control is transferred to the statement specified in the clause. If there is no appropriate exception-recovery clause, a request is made that the current input reply be reentered. In both cases, the request is made in the form of a warning message. The user is reprompted with a question mark. (The question mark is used even if a prompt is specified.)

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.

For more information, see "Input/Output Data Rules" on page 86.

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 remain unchanged. Also, if the beginning of the input reply starts with a comma, the first item of the input list will be passed over and remains unchanged. Consecutive commas may be separated by spaces.

If the last character of the input reply is a comma, additional input is expected on the next line. Each input line must be less than or equal to 156 characters in length. 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 remain 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 values corresponding to the elements in the array are accepted and assigned to elements 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.

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, a message will be displayed and the data will be ignored. If SKEY is set to IGNORE, pressing the PF key has no effect. If SKEY is set to GOTO, control is transferred to the specified line (see "ON Condition Statement" on page 298).

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; *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.

In the input reply, the notation:

```
j*
```

where j is a nonzero, unsigned integer constant, indicates j implicit data entries. This has the effect of passing over the next j items in the input list, thereby leaving them unchanged.

Given the following INPUT statement:

```
100 INPUT A, B, C, D, E
```

and a user response of:

```
10,3*,11
```

the variables A, B, C, D, and E would appear as follows before and after execution of the INPUT statement:

```
            A   B   C   D   E
BEFORE      1   2   3   4   5
AFTER      10   2   3   4  11
```

Character constants in the input reply are normally enclosed in quotation marks. However, these quotation marks may be omitted for character constants which:

- Contain no commas
- Have no leading or trailing spaces
- Contain no leading or trailing quotes
- Do not start with a nonzero, unsigned integer followed by an asterisk
- Are not null strings

If the input reply is null (or consists only of spaces) and corresponds to a character variable which is last in the input list, then that character variable will be set to the null string.

Numeric data is rounded to 17 digits for decimal data, 6 hexadecimal digits for real single, 14 hexadecimal digits for real double, and to an integer value for integer data. See "Numeric Data Types" on page 26 for more information.

Exception Conditions: A numeric entry causes an overflow (OFLOW) condition. An underflow (UFLOW) condition occurs when the numeric entry is too small to be represented and the numeric variable is not integer. If a numeric value cannot be converted as required, the CONV exception occurs.

If a numeric value is too large for the receiving variable, an overflow exception (OFLOW) occurs.

If a numeric value is too small to be represented by a receiving decimal or real variable, an underflow exception (UFLOW) occurs.

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 exception (SOFLOW) occurs.

The SYSTEM action for the above exception (except UFLOW) is to output a message and request that that line be reentered. For an UFLOW exception, the SYSTEM action is to display a message and use zero for the value.

If there are no more values in the file (when in batch) to assign to remaining input list items, an end-of-file exception (EOF) occurs.

You may cause an attention interrupt (see your IBM BASIC System Services manual for instructions on how to do this) while BASIC is waiting for input. You must still satisfy the input, however, before the ATTN exception condition is generated. The easiest way to do this is to enter a slash (/) after causing the interrupt. If the appropriate exception-recovery clauses are included on the INPUT statement, or in a specified EXIT statement, the action taken is determined by the specified exception-recovery routine.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128. See also "EXIT Statement" on page 197.

Example

```
100 INPUT "ENTER NAME:":NAME$
200 INPUT "ENTER HOME & BUSINESS PHONE:":HOME$,BUS$
```

The above statements will prompt for a name, then home and business phone numbers.

Example

```
100 DIM ENTRY$*39
110 ENTRY$="ENTER NAME:ENTER HOME & BUSINESS PHONE:"
.
.
.
120 INPUT PROMPT ENTRY$(1:11):NAME$
130 INPUT PROMPT ENTRY$(12:39): HOME$,BUS$
```

The above example uses PROMPT with a character expression to prompt for a name, then home and business phone numbers.

INPUT FIELDS Statement

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 is zero.

field-definition
is one of the following:

```
character-expression

MAT character-array-name
```

and where:

character-expression *or each* **character-array-element**
must evaluate to:

```
row, column, data-form [,[leading] [,trailing]]
```

and where:

row
is an unsigned, nonzero integer, specifying the screen row of the field.

column
is an unsigned, nonzero integer, specifying the column of the first character in the field.

data-form
is one of the data-forms shown in Figure 37 on page 250.

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 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 is specified, it overrides both H and N; H overrides N. N is the default.

Leading control attributes that have meaning to 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 nonattribute character after the field.

C
position the cursor to this field first

The trailing control attribute that has meaning to 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 nonattribute character after the field.

input-list
is a list of one or more variables, array elements, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses :

```
EXIT line-reference
CONV line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

Data-Form	Meaning
w	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.
C [w]	Character data padded with spaces on input.
V [w]	Character data with trailing spaces 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 spaces.

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 37. INPUT FIELDS Statement—Data-Forms

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 nonattribute 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.)

Nulls are removed from the values in the input fields when the ENTER key or PFkey is pressed.

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. (The first field definition corresponds to the first input list item, the second field definition corresponds to the second input list item, and so forth.)

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 88.)

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 display a message and 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 unsigned, nonzero integers that specify the starting location of each field. Row 1, column 1 is the upper left 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 37 on page 250 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. See "FORM Statement" on page 205.

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 more information, see "Input/Output Data Rules" on page 86.

Exception 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 (see "EXIT Statement" on page 197).

The I/O exception conditions interact with the ON Condition statement as described in "Exception Handling in I/O Statements" on page 128.

Example

```
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

```
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 N is in effect.

Normally, pressing a PF key during the execution of an INPUT statement causes an SKEY exception; see "ON Condition Statement" on page 298. Arising from the INPUT FIELDS statement, the SKEY condition causes the same result regardless of whether SYSTEM or IGNORE action was specified. Thus, the only result is that processing continues normally.

For the SKEY condition with the GOTO action, no data is transferred, but data entered remains on the screen.

Example

```
100 OPTION BASE 1
110 DIM A$(4), B$(4), NAME$*30, ADDR$*30, CITY$*30,&
  & ST_ADDR_CODE$*30
130 DATA "10,20,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 to be filled before ENTER is pressed. Values of fields that are unchanged will be assigned to the corresponding variable or array element.

Statements 200 through 220 serve as a check that the user hasn't pressed ENTER accidentally. If the user wants 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, or from the terminal.

Format

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 an unsigned nonzero integer within the range 0 to 255, and which identifies the file to be read.

input-prompt
is one of the following:

```
PROMPT character-expression
PROMPT quoted-character-string
quoted-character-string
```

Note: The *#fileref* and *input-prompt* may occur in either order.

input-list
is a list of one or more variables, array elements, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
EOF line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

INPUT File Statement

Description

The INPUT File statement, when referring to a file opened as a DISPLAY type file, retrieves data similar to the way the INPUT statement accepts data from a terminal: a record from the file is processed as if it was a line entered on the terminal in response to an INPUT request. However, there are some differences:

- During input from a file, the input prompt has no meaning and is ignored (if the fileref is 0, input comes from the terminal, not a file).

- 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:

  - An attempt is made to read a character data item into a numeric variable.
  - The number of data values does not match the number of variables in the input list.
  - 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 243.)

When operating on a file opened with STREAM organization, the INPUT File statement acts like a GET statement. (See "GET Statement" on page 225.)

When operating on a file opened with INTERNAL type and SEQUENTIAL organization, the INPUT File statement acts like a READ File statement. (See "READ File Statement" on page 346).

The INPUT File statement may *not* be used with files opened with NATIVE type.

For more information on files see "BASIC File Capabilities" on page 61.

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 85 for more information.

Fileref: The fileref must refer to a display or internal file. (See "Combinations of File Type, Organization, Access, and Records" on page 64.) The internal file may have either sequential or stream organization.

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 sequential 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 avoiding the CONV condition.

For more information, see "Input/Output Data Rules" on page 86.

Exception Conditions: If the fileref is 0, see "INPUT Statement" on page 243 for details on exception conditions.

If a numeric value is too large for the receiving variable, an overflow exception (OFLOW) occurs.

If a numeric value is too small to be represented by a receiving decimal or real variable, an underflow exception (UFLOW) occurs.

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 exception (SOFLOW) occurs.

If there are no more values in the file to assign to remaining input list items, an end-of-file exception (EOF) occurs.

For an internal sequential file, a conversion exception (CONV) occurs if all the items in the input list have been satisfied, but more values exist in the current record. Also, if a numeric value cannot be converted as required (for example, if you try to input character data into a numeric variable), the CONV exception occurs.

The string overflow (SOFLOW), conversion (CONV), and end-of-file (EOF) conditions described above, as well as the input/output exception condition (IOERR), may be recoverable if the corresponding exception-recovery clauses are included on the INPUT File statement, or by specifying an EXIT clause.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128. See also "EXIT Statement" on page 197.

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)} [,{identifier|(letter-list)}]...]
```

where:

identifier
is a 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.

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 user-defined function names.

An identifier explicitly stated in an INTEGER statement may end with the self-typing character % but not with # or $.

If you specify a *letter-list*, 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, DEFDBL, DEFSNG, or REAL statement by identifier.

The letter list may be specified as either single letters, such as (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.

Example

```
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 or real and EDGAR$ assigned character.

Immediate Execution

You can use the INTEGER statement to set the type of immediate variables and arrays. The format and description are the same as for an INTEGER statement in a program.

See "Immediate Statements" on page 133 and "Immediate Type and Dimensions" on page 135 for the rules regarding the interaction 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 one of the following:

```
numeric variable
character variable (with optional substring notation)
subscripted numeric array element
subscripted character array element (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 equal sign is evaluated and assigned to the numeric or character variable to the left of the equal 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 equal 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 and right side may have different numeric types.

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, real single, real double, 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 equal sign, the result of the expression is converted to the type of the variable on the left. Rounding occurs when converting. Conversion to integer is the same as

```
variable = IFIX(expression)
```

See "IFIX(X) Rounded Integer Value" on page 395. Conversion to real single is the same as

```
variable = SNG(expression)
```

See "SNG(X) Real Single" on page 412. Conversion to real double is the same as

```
variable = DBL(expression)
```

See "DBL(X) Real Double" on page 388. Conversion to decimal is the same as

```
variable = DEC(expression)
```

See "DEC(X) Decimal" on page 389.

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 to 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 first variable in the immediate LET statement is a variable having the same name as one of the BASIC commands, and the variable name is not followed by an equal sign.

Example

```
LIST = 3            is accepted
LET LIST = 3        is accepted
B, CHANGE, A = 9    is accepted
RUN(3) = 6          is an error
LET RUN(3) = 6      is accepted
SAVE, B = 3         is an error
```

For additional information on immediate execution, see "Immediate Statements" on page 133.

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
is one of the following:

```
PROMPT character-expression
PROMPT quoted-character-string
quoted-character-string
```

input-list
is a list of one or more character variables, array elements, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

Description

LINE INPUT may also be spelled LINPUT.

Inside the BASIC environment, 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 your IBM BASIC System Services manual.

Character strings which contain commas, spaces, 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.

For more information, see "Input/Output Lists" on page 85.

Input-list: For more information, see "Input/Output Data Rules" on page 86.

Input-Prompt: If the statement specifies an input prompt, the result of the PROMPT character 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 LINE INPUT statement causes any prompt or the question mark to be appended to the data from the last PRINT statement and 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 character 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.

Exception Conditions: The exception conditions IOERR (input/output exception) and SOFLOW (string overflow) may be recoverable if an exception-recovery 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 exception-recovery conditions interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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$ with a length of 11. BASIC then prompts (?) for the second value to be stored in B$.

LINE INPUT/LINPUT File Statement

The LINE INPUT File statement reads unformatted data from a display 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 an integer within the range 0 to 255, and which identifies the file to be processed.

input-prompt
is one of the following:

```
PROMPT character-expression
PROMPT quoted-character-string
quoted-character-string
```

Note: The *#fileref* and *input-prompt* clauses may occur in any order.

input-list
is a list of one or more character variables, array elements, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
EOF line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

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, spaces, and other characters usually thought of as delimiters, is assigned to each successive character string variable in the input list.

For more information on files, see "BASIC File Capabilities" on page 61.

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.

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.

For more information, see "Input/Output Lists" on page 85.

Input-list: For more information, see "Input/Output Data Rules" on page 86.

Fileref: A file which is accessed with this statement must have display type. The length of the records must not exceed the length of the corresponding character variables in the input list. (See "Combinations of File Type, Organization, Access, and Records" on page 64.)

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 263).

Exception Conditions: The EXIT, IOERR, and SOFLOW clauses function as they do for a terminal. In addition, the EOF 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 exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 described in "Logical Expressions" on page 57.

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 190, "Loop Control Statements" on page 74, and "EXIT IF Statement" on page 200.

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. At least one must appear when using the second format.

Description

The MARGIN statement is an executable statement that 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.

The effects of this statement remain until another MARGIN or MARGIN File statement explicitly changes one or more margins of the terminal and, in that case, only the explicitly modified margins are affected. When program execution is started, the margins are reset to the initial default values.

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 (unless the right margin is 0).

If no left margin is specified, the left margin is not changed.

Initially, the left margin is 1.

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 right margin of 0 is specified, the terminal's line width is used.

If no right margin is specified, the right margin is not changed.

Initially, the right margin is equal to the width of the terminal.

For PRINT statements without IMAGE or FORM control, the right margin, if not zero, must exceed the left margin by an amount equal to the print zone size (PRTZO). See "OPTION Statement" on page 312.

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 top margin is not changed.

Initially, the top margin is 1.

MARGIN BOTTOM: The value specified for the bottom margin must be within the range 0 to 32767.

A bottom margin of 0 implies no line limit, and no ENDPAGE (or PAGEOFLOW) condition is to be generated.

If no bottom margin is specified, the bottom margin is not changed.

Initially, the bottom margin is 0.

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 hard copy 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, the OPTION PRTZO statement, or the SET OPTION PRTZO command, 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, and so forth.

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

The form begins at the LEFT margin.

If the position of a value to be displayed will start beyond the right margin, the current line is written, but 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, starting at the left margin.

If a POS control specification specifies a position between 1 and left margin -1, inclusive, an 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 ensure 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 22 print lines have scrolled onto the screen and before displaying the 23rd line, the scrolling stops and the bottom line 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 23rd line scrolls onto the bottom line of the screen and scrolling continues.

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 298), the ENDPAGE exception is generated immediately, regardless of whether 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 ENDPAGE 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.

Note that a subsequent print will continue printing without causing an ENDPAGE condition. You must cause a NEWPAGE to start a new page before counting of the lines resumes. This allows for printing running headings.

Example

```
100 ON ENDPAGE GOTO FOOTING_HEADING
.
.
.
200 FOOTING_HEADING: PRINT
210 PRINT RUNNING_FOOT$
220 PRINT NEWPAGE
230 PRINT RUNNING_HEAD$; TAB(50); DATE$
240 PRINT
250 CONTINUE
.
.
.
```

MARGIN File Statement

The MARGIN File statement specifies the page margins for display type 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 an unsigned 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 when using the second format.

Description

The MARGIN File statement functions for a display type 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 changes one or more margins of that file, or until the file is closed. Only those margins that are explicitly changed are affected.

The first format may be used to set the right margin.

Fileref: The fileref must refer to a display file. (See "Combinations of File Type, Organization, Access, and Records" on page 64.)

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).

If no left margin is specified, the left margin is unchanged.

Initially, the left margin is 1.

MARGIN RIGHT: The value specified for the right margin must be within the range 0 to the record length of the file.

If a right margin of 0 is specified, the record length of the file is used.

If no right margin is specified, the right margin is not changed.

Initially, the right margin is the lesser of 133 and the record length.

MARGIN TOP: The value specified for the top margin must be within the range 0 to the bottom margin (unless the bottom margin is 0).

If no top margin is specified, the top margin is unchanged.

Initially, the top margin is 1.

If 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 value specified for the bottom margin must be within the range 1 to 32767.

A bottom margin of 0 implies no line limit, and no ENDPAGE condition is generated.

If no bottom margin is specified, the bottom margin is not changed.

Initially, the bottom margin is determined by your installation. The IBM-distributed default is 60.

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 269.)

If fileref is zero, the MARGIN File statement acts as a MARGIN statement and sets the margins for the terminal.

Example

This example creates a file that contains 450 slashes in a 15 character by 30 character area. This 15x30 "box" will be placed 10 lines from the top of the page and 20 spaces from the left margin.

```
100 OPEN #1: 'SLASH$', TYPE DISPLAY, OUTPUT
110 MARGIN #1 RIGHT 49 LEFT 20 TOP 10 BOTTOM 24
120 FOR X = 1 TO 450
130   PRINT #1: '/';
140 NEXT X
150 CLOSE #1
160 END
```

This example creates an output file, called SLASH$. In order to see what is in it, you have to go back to the system and print it out.

It will look like this:

```
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
//////////////////////////////
```

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)
array-function
intrinsic-function
```

and where:

arrayname1 *and* **arrayname2**
are arraynames (see Description for restrictions on type and dimensions).

expression
is a numeric or character expression (see Description for restrictions on type).

array-function
is one of the following:

```
AIDX (arrayname1)
ASORT (arrayname1)
[(expression) *] CON
DIDX (arrayname1)
DSORT (arrayname1)
function[(arg [,arg]...)]
[(expression) *] IDN
INV (arrayname1)
NUL$
TRN (arrayname1)
ZER
```

intrinsic-function
is the name of an intrinsic function (other than DOT, PRD, SIZE, SRCH, SUM, or UDIM; DET may not be used with an argument).

arg

is an array or a scalar argument. The number and type of arguments must be as defined for the intrinsic function.

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 equal sign.

When the array is a character array, the character value 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 exception occurs.

All array statements which result in assignment (with the exception of AIDX and DIDX when the argument is a character array) 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

```
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, 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, real single, real double 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. See "LET (Scalar Assignment) Statement" on page 260.

Redimension Clause: When the redimension clause is specified, the array is redimensioned dynamically; the number of dimensions and the size in each dimension are 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

```
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 BASE option 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), provided the total number of elements for the redimensioned array does not exceed the total number of elements specified by the array's original dimensions. This means that, for array B in the above example, you can change any one of the dimensions or even add dimensions, provided the total number of elements doesn't exceed 20 (when the BASE 0 option is in effect) or 12 (when the BASE 1 option is in effect). See "Redimensioning" on page 41.

Simple 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 array, *arrayname1* must be a character 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. The value of the expression is evaluated once.

If *arrayname* is a character array, the expression must be a character 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.

Example

```
200 MAT A = (PI/2)
```

This statement 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 51.

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 element of an array named ARAZ is to be given the sum of the corresponding elements 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), and so forth, until all 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 38 on page 280.

If the statement is changed to subtract, by changing line 110 to:

```
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 38.

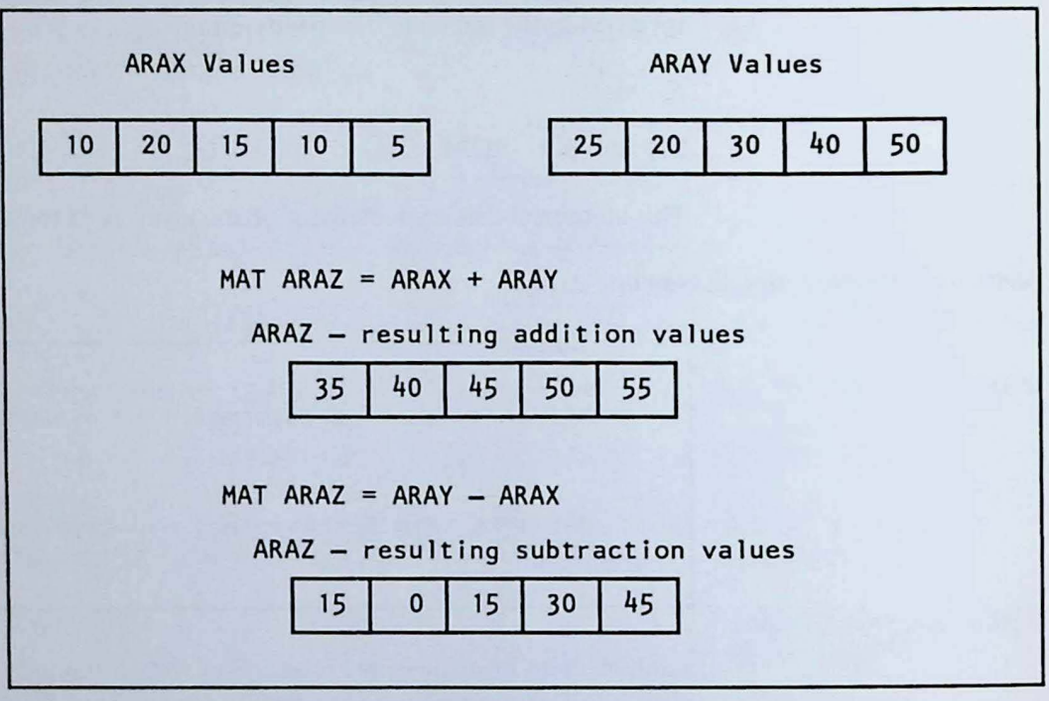


Figure 38. MAT Statement—Addition and Subtraction Example

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 subscript indicates 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 contains the values 2, 3, 4, 5, 6, and 7, and ARAY contains the values 8, 9, 10, 11, 12, and 13, and that you want to 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 will be as shown in Figure 39.

ARAX Values

2	3
4	5
6	7

ARAY Values

8	9	10
11	12	13

MAT ARAZ = ARAX * ARAY

Resulting ARAZ Values

2*8+3*11 = 49	2*9+3*12 = 54	2*10+3*13 = 59
4*8+5*11 = 87	4*9+5*12 = 96	4*10+5*13 = 105
6*8+7*11 = 125	6*9+7*12 = 138	6*10+7*13 = 151

Figure 39. MAT Statement—Matrix Multiplication Example

The MAT statement illustrated in Figure 39 on page 281 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 in which each element of an array (*arrayname1*) is multiplied by the scalar result of *expression* (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:

1. 2+2 is evaluated, giving 4,
2. ARAX is redimensioned to the same dimension as ARAY, and,
3. Each element of ARAX receives the product of 4 times the corresponding element of ARAY.

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 40.

ARA$ Values

abc	def
ghi	jkl

ARB$ Values

www	xxx
yyy	zzz

```
MAT ARC$ = ARA$ & ARB$
```

Resulting ARC$ Values

abcwww	defxxx
ghiyyy	jklzzz

Figure 40. MAT Statement—Matrix Concatenation Example

Scalar Concatenation in Character Arrays

Format

```
MAT arrayname = (expression) & arrayname1 [(redimension)]
        or
MAT arrayname = arrayname1 & (expression) [(redimension)]
```

The value of *expression* is concatenated to (joined together with) each element of *arrayname1* and assigned to the corresponding element of *arrayname*.

If *(expression)* precedes *arrayname1*, the value of *(expression)* is concatenated on the left.

If *(expression)* follows *arrayname1*, the value *(expression)* is concatenated on the right.

Arrayname and *arrayname1* 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 *arrayname1*.

If the redimension specification is given, *arrayname* is dynamically redimensioned to the values specified in the redimension specification.

Suppose 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 41.

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 41.

ARA$ Values

abc	def
ghi	jkl

DATA$

mmm

MAT ARC$ = (DATA$) & ARA$

Resulting ARC$ Values

mmmabc	mmmdef
mmmghi	mmmjkl

MAT ARC$ = ARA$ & (DATA$)

Resulting ARC$ Values

abcmmm	defmmm
ghimmm	jkimmm

Figure 41. 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.

Scalar Function References

Format

```
MAT arrayname = function [(arg [,arg]...) ] [(redimension)]
```

Function is the name of an intrinsic function and *arg* is an argument to the function and is either a scalar expression or an array.

Any intrinsic function may be used except DOT, PRD, SIZE, SRCH, SUM, and UDIM. DET may not be used with an argument.

The function is invoked separately for each element of arrayname. The argument determines the value to be used for each invocation of the function. If an argument is a scalar expression, then the same value (the value of that expression) is used as the argument in determining every element of *arrayname*.

Example

```
10 DIM A$(2,2)
20 X$ = "        ABC"
30 MAT A$ = LTRM$(X$)
```

results in each element of A$ being assigned the value "ABC."

If an argument is an array, then the argument to be used by the function in determining an element of *arrayname* is taken from the corresponding element of the argument array. If more than one array appears in the argument list, they must all have the same number of dimensions and the same size in each dimension.

If a simple name argument has not been previously defined, it defaults to a scalar.

This capability is useful when calling FORTRAN or COBOL programs with character array arguments (to pad or trim off spaces for all the elements of an array using RPAD$ and RTRM$ functions applied to the whole array).

Example

```
10 OPTION BASE 1
20 DIM A$(2,2), B$(2,2), C(2,2)
30 C(1,1)=1: C(1,2)=2: C(2,1)=3: C(2,2)=4
.
.
.
140 MAT A$ = LWRC$(B$)
```

results in the elements of A$ assigned as follows:

```
A$(1,1) = LWRC$(B$(1,1))
A$(1,2) = LWRC$(B$(1,2))
A$(2,1) = LWRC$(B$(2,1))
A$(2,2) = LWRC$(B$(2,2))

140 MAT A$ = RPT$("ABC",C)
```

results in the elements of A$ assigned as follows:

```
A$(1,1) = RPT$("ABC",1) = "ABC"
A$(1,2) = RPT$("ABC",2) = "ABCABC"
A$(2,1) = RPT$("ABC",3) = "ABCABCABC"
A$(2,2) = RP$$("ABC",4) = "ABCABCABCABC"
```

The function is reevaluated for each element assignment.

Example

```
10 MAT A$ = TIME$
```

may result in different times for each element of A$.

If no redimension specification is given, *arrayname* is dynamically redimensioned to the dimensions of any argument array. If a redimension specification is given, *arrayname* is dynamically redimensioned to the specified values.

If there are no array arguments and no redimensioning, the result array has the same dimensions as the original array to the left of the equal sign.

Example

```
10 MAT A = RND
```

assigns pseudorandom numbers to the elements of array A.

```
20 MAT B = RND(0.5)
```

assigns the same pseudorandom number to every element of B.

```
30 MAT C = RND(A)
```

Assigns a new pseudorandom number to each element of C using a different seed value from array A.

Note: When an array is not used as one of the function arguments and the same value is desired for all elements on the left of the equal sign, put parentheses around the function for better performance (the function is evaluated only once instead of once for each element). See "Scalar Assignment" on page 278.

For example, if the same time is desired in each element, use the following statement:

```
10 MAT A$ = (TIME$)
```

Array Functions

Array functions are a collection of predefined functions that produce array results. Although they are similar to the (scalar) intrinsic functions in many ways, there are significant differences.

- They return an array result.
- They take at most one argument, while some intrinsic functions have two or more.
- Many array functions can return either a character or numeric array result, depending on the type of their argument. Note that such functions can return a character array even though the function name does not end with a dollar sign. However, array functions whose names end with a dollar sign can return only character arrays.

- They can be invoked only within the MAT assignment statement and may be used only with a relatively narrow set of syntactical constructs. For example, array functions cannot be used:
  - within expressions
  - as arguments to intrinsic functions, or
  - in the output list of PRINT statements.

The following sections describe the various array functions that may be used in a MAT assignment statement.

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 either a numeric or a character array.

Arrayname1 must be a one-dimensional array (vector).

The resulting vector is integer.

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 BASE option affects the results of the AIDX function. Note that the subscripts of the indexed array are the results of the AIDX function; therefore the BASE option setting affects the results obtained.

Character arrays are indexed according to the collating sequence specified. Therefore, if *arrayname1* is a character array, the COLLATE option setting affects the results. See "OPTION Statement" on page 312.

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
```

Ascending Sort Array Function (ASORT)

Format

```
MAT arrayname = ASORT (arrayname1) [(redimension)]
```

The ASORT function sorts character or numeric arrays in ascending sequence. The array being sorted is *arrayname1*.

Character arrays are sorted based on the COLLATE option selected. See "OPTION Statement" on page 312.

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

```
 90 OPTION COLLATE NATIVE
100 DIM ASC$(3,3),ASB$(3,3)
110 MAT ASB$ = ASORT(ASC$)
```

Since OPTION COLLATE NATIVE is in effect, the elements of array ASC$ are sorted according to the EBCDIC collating sequence and stored in ascending order in array ASB$.

Note: The EBCDIC collating sequence can be changed to the ASCII collating sequence if OPTION COLLATE STANDARD is specified (see "OPTION Statement" on page 312).

Example

```
100 DIM AA(100),BB(100)
110 MAT AA = ASORT(BB)
```

The elements 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 (decimal, integer, real) and the result is then converted to the type of array to the left of the equal sign.

Constant Array Function (CON)

Format

```
MAT arrayname = [(expression)*] CON [(redimension)]
```

The CON function sets the value of each element to the value of *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 1s, 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 12.

Note: The two statements below give you the same results (see "Scalar Assignment" on page 278).

```
110 MAT ARAX = (12)*CON
110 MAT ARAX = (12)
```

Descending Index Array Function (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-dimensional array with the same number of elements as *arrayname1*.

The resulting vector is integer.

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 BASE option affects the results of the DIDX function. Note that the subscripts of the indexed array are the results of the DIDX function; therefore, the BASE option affects the results obtained.

Character arrays are indexed according to the collating sequence specified. Therefore, the OPTION COLLATE setting affects the results. See "OPTION Statement" on page 312.

Example

The following statements index the elements of character array (ARA$) and assigns them to ARB% in descending sequence (the default BASE 0 option 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 ARA$ 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
```

Descending Sort Array Function (DSORT)

Format

```
MAT arrayname = DSORT (arrayname1) [(redimension)]
```

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 312.

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

```
110 OPTION COLLATE NATIVE
120 DIM DEC$ (3,3), DEB$ (3,3)
130 MAT DEB$ = DSORT(DEC$)
```

Since OPTION COLLATE NATIVE is in effect, the elements 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 (see "OPTION Statement" on page 312).

Example

```
100 DIM AA(100),BB(100)
110 MAT AA = DSORT(BB)
```

The elements of array BB are sorted in descending numeric sequence and stored in descending order in array AA.

For numeric arguments, the same type argument is returned (decimal, integer, real) and the result is then converted to the type of array to the left of the equal sign.

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 (*expression*) is used with the identity function, the value of the expression is assigned to each diagonal array element (one whose subscripts are equal), and zero is assigned to all other array elements.

Example

The following statements assign the value 1 to ARAX(1,1), ARAX(2,2), ARAX(3,3); all other array elements are assigned the value 0.

```
 50 OPTION BASE 1
100 DIM ARAX(3,3)
110 MAT ARAX = IDN
```

Execution results are shown in Figure 42.

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 elements are assigned a value of zero.

```
50 OPTION BASE 0
100 DIM AA(3,3)
110 MAT AA = (8)*IDN
```

Execution results are shown in Figure 42.

MAT ARAX = IDN

Values in ARAX(3,3)

1	0	0
0	1	0
0	0	1

MAT AA = (8)*IDN

Values in AA(3,3)

8	0	0	0
0	8	0	0
0	0	8	0
0	0	0	8

Figure 42. MAT Statement—IDN Function Examples

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 because of roundoff and precision restrictions.

The array on the right *(arrayname1) must* be numeric, two-dimensional and square (the number of rows must equal the number of columns).

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)] Determinant" on page 389.)

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 43.

```
MAT ARAJ = INV(ARAK)

ARAK Values          Resulting ARAJ Values
 1   1                2   -1
 1   2               -1    1
```

Figure 43. MAT Statement—INV Function Example

INV accepts integer arrays as arguments but always produces decimal results if ARITHMETIC DECIMAL is in effect and real results if ARITHMETIC NATIVE is in effect. The real results are real single or real double, depending on whether

SPREC or LPREC is in effect. If the array argument is decimal or real, the result is of the same type.

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.

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.

Example

```
100 DIM ARAA$ (2,2)
110 MAT ARAA$=NUL$
```

After the MAT statement is executed, ARAA$ is still a two-dimensional array, and each element is a null character string. Note that this yields the same results as:

```
100 DIM ARAA$(2,2)
110 MAT ARAA$=("")
```

Transpose Array Function (TRN)

Format

```
MAT arrayname = TRN (arrayname1) [(redimension)]
```

The function TRN transposes an array.

Arrayname and *arrayname1* must both be numeric arrays or must both be character arrays, with up to seven dimensions.

Arrayname will be dynamically redimensioned to the same number of dimensions as *arrayname1,* but the bounds for the dimensions are in the reverse order. An element of *arrayname* has the same value as in arrayname1 except with the subscripts reversed; *arrayname* (i_1,..., i_n) is equal to *arrayname1* (i_n,..., i_1).

Note that if *arrayname1* has only one dimension then TRN *(arrayname1)* is equal to *arrayname1*.

For a two-dimensional array, the above rule would cause values contained in column 1 of *arrayname1* to be transferred into row 1 of *arrayname*, the values in column 2 to be transferred into row 2, and so forth.

After the assignment, if a redimension specification is given, *arrayname* is dynamically redimensioned to the value specified in the redimension specification.

TRN always produces an array of the same type as the original.

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 44.

If ARAX contained the values:

1	10	100	1000
2	20	200	2000
3	30	300	3000

MAT ARAY = TRN(ARAX)

ARAY would contain:

1	2	3
10	20	30
100	200	300
1000	2000	3000

Figure 44. MAT Statement—TRN function Example

This function is useful when you want to call a subprogram in FORTRAN (for example) and pass an array. See "CALL COBOL, FORTRAN or PLI Statement" on page 152.

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 45.

```
MAT ARAX=ZER

ARAX
```

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Figure 45. MAT Statement—ZER Function Example

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 133 gives additional information.

NEXT Statement

The NEXT statement is the end delimiter of a FOR loop.

Format

```
NEXT variable
```

where:

variable
is a simple 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 74 and "FOR Statement" on page 203.

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 one of the following:

```
IGNORE
GOTO line-reference
SYSTEM
```

line-reference
is a line number or line label.

Spaces may appear between GO and TO in the keyword GOTO.

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. See your IBM BASIC System Services manual for information on how to cause an attention interrupt.

CONV An exception has occurred during an input/output operation. The exception can be a numeric data conversion exception, or can be caused by 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 269 and "MARGIN File Statement" on page 273.

ERROR This is a generalized exception; it applies to any exception 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 This is 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 intrinsic function KEYNUM returns the number of the PF key pressed.

SOFLOW A string overflow condition has occurred; 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 or real value allowed, depending on the type of the expression.

ZDIV The condition of division by zero of numeric data has occurred. Zero raised to a negative power, INV of a singular matrix, and MOD and REM with a zero as the second argument also produce an exception that is classed as a ZDIV exception.

See Appendix A, "Exception Codes" on page 493 for a list of the exception codes and the corresponding ON condition.

ON Condition Actions: You may pick one of three *actions* to occur when the specified condition happens, as follows:

IGNORE Allows processing to continue normally.

GOTO line-reference
Causes the indicated transfer of control to a line label or line number.

SYSTEM Permits a predetermined system-controlled response to the condition.

Exception-recovery clauses referring to the same conditions as those in the ON override the ON Condition action when an exception occurs for a statement with that exception-recovery clause.

Whenever an exception causes a transfer of control, the exception code is available by using the intrinsic function ERR. The line number of the statement where the exception occurred is available by using the intrinsic function LINE. In addition,

when an SKEY exception occurs, the intrinsic function KEYNUM returns the number of the PF key which was pressed.

Figure 46 indicates which of the actions may be used with a particular condition and what they mean.

ON	IGNORE	GO TO line-reference	SYSTEM
ATTN	No user message. Continue [4].	No user message. Transfer control.	Batch execution: ignore. Interactive execution: stop at the next statement. User message [3].
CONV	Not allowed.	No user message. No data transfer. Transfer control.	User message. Stop [1].
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 269 and "MARGIN File Statement" on page 273.
ERROR	Not allowed.	No user message. No data transfer. Transfer control.	User message. Stop [1,5].
OFLOW	No user message. Replace with signed INF. See "INF Infinity" on page 396 Continue.	No user message. No data transfer. Transfer control.	User Message. Replace with signed INF. See "INF Infinity" on page 396 Continue [1].
SKEY	No user message. Continue.	No user message. No data transfer. Transfer control.	No user message. Continue [2].
SOFLOW	No user message. Excess data is truncated on the right. Continue.	No user message. No data transfer. Transfer Control.	User message. Stop [1].
UFLOW	No user message. Replace with zero. Continue.	No user message. No data transfer. Transfer control.	User message. Replace with zero. Continue.

Figure 46 (Part 1 of 2). ON Conditions and Their Actions

ON	IGNORE	GO TO line-reference	SYSTEM
ZDIV	No user message. Replace with signed INF. See "INF Infinity" on page 396 Continue.	No user message. No data transfer. Transfer Control.	User message. Replace with signed INF. See "INF Infinity" on page 396 Continue.

Figure 46 (Part 2 of 2). ON Conditions and Their Actions

Notes to Figure 46 on page 300 :

1 If the exception occurs on INPUT from the terminal, the SYSTEM action is to display a user message and to request that the entire input reply be reentered. See "INPUT Statement" on page 243.

2 If the exception occurs on INPUT or LINE INPUT from the terminal, the SYSTEM action is to ignore only the last line of input, display a user message, and continue the input from the terminal.

3 Inside the BASIC environment, execution may be continued or terminated as in any other program interruption (see "RUN Command" on page 472 and "GO Command" on page 445). Outside the BASIC environment, program execution may be continued by pressing the ENTER key or entering GO; execution may be terminated by entering QUIT.

4 A source program with ON ATTN IGNORE as its first executable statement will, when compiled, produce more compact, efficient code than a program without the statement.

5 If index in TAB is less than one, one is used and execution continues. If the argument of POS is less than the left margin, the left margin is used and execution continues. If the fileref is zero in an OPEN, CLOSE, RESET, or SCRATCH, the statement is ignored and execution continues. If NEWPAGE is requested for a non-printer file, no eject is performed and execution continues.

Example

```
 * LIST
100 on soflow goto toolong
110 dim name$*8
120 input " Please enter the name of your dog: ": name$
130 print " What a nice name for a dog!"
140 goto fin
150 toolong: print "The name is too long to fit on a dog license."
160 print "You should think of a shorter one."
170 retry
180 fin: print "The name was accepted."
190 end
 * RUN
 Please enter the name of your dog:  Roverover
The name is too long to fit on a dog license.
You should think of a shorter one.
 Please enter the name of your dog:  Rover
 What a nice name for a dog!
The name was accepted.
END AT LINE 190.
```

ON GOTO/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} line-reference
    [,line-reference]... [NONE line-reference | ELSE statement]
```

where:

numeric-expression
can be any numeric expression as described in "Numeric Expressions" on page 47.

line-reference
is a line number or line label.

statement
is an imperative statement. See Figure 34 on page 233 for a listing of imperative statements.

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 numeric expression is evaluated, and its rounded integer value determines the element of the line reference list to which the GOTO or GOSUB statement branches.

Example

```
100 ON ABLE GOTO ONE, TWO, THREE
```

If ABLE equals 1, the program branches to ONE; if ABLE equals 2, the program branches to TWO; if ABLE equals 3, the program branches to THREE.

If the rounded integer value of the expression is less than one or greater than the number of line numbers and labels in the list, an exception 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 less than one or greater than the number of line numbers and labels 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 does not designate a line number or label in the list.

Example

```
100 ON I GOTO ONE,TWO NONE NONE_OF_THESE
```

If I equals any value less than one or greater than two, the program branches to the line labelled NONE_OF_THESE.

If the *line-reference* is a nonexecutable statement, control is passed to the first executable statement following the specified statement.

Transfer to a nonexistent line reference results in an exception.

If ELSE is specified and the value of the expression does not designate a line number or label in the list, the imperative statement is executed. That statement may be any of those shown in Figure 34 on page 233.

Example

```
100 ON A+B GOTO ONE,TWO 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: branches to a 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 or line labels represent the first statement of subroutines 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 multi-statement 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 73.

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 the rounded integer value of ACTION is	Control is transferred to
1	DEBIT
2	CREDIT
3	CURRENT_BALANCE

If the rounded integer value of ACTION is any other value, control is transferred to line 500.

OPEN Statement

The OPEN statement activates a file and specifies file attributes.

Format

```
OPEN #fileref: [NAME] file-id [file-attributes] [err]
```

where:

fileref
is a numeric expression which, when evaluated and rounded, must be an unsigned integer within the range 1 to 255. It is a number you assign to an external file to identify that file wherever it is referred to in a program unit.

file-id
is a character expression, the value of which contains all of the information necessary to define the file to the system. The value of the character expression must evaluate to:

```
file-spec [, DEVICE {PRINTER|3800}]...
```

For valid entries for file-spec, see your IBM BASIC System Services manual. A simple file-spec of eight characters or less beginning with a letter and consisting of letters and digits is supported by both VM and MVS.

If the file is a display output file that is to include a carriage control character at the beginning of each record, you should specify the DEVICE PRINTER clause as part of the value of the *file-id*.

If the file is a display output file to be listed on a 3800 printer, you should specify the DEVICE 3800 clause as part of the value of the *file-id*. This clause 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. For more information, see *IBM 3800 Printing Subsystem Programmer's Guide*.

file-attributes
describe in any order: file access, type, organization, position, and record type. Certain restrictions on combinations and values of attributes are system dependent. For details and restrictions, see your IBM BASIC System Services manual. The attributes are:

ACCESS
ORGANIZATION
POINTER
RECORDS
TYPE

These attributes are described below.

err

is one of the following exception-recovery clauses:

```
EXIT line-reference
IOERR line-reference
```

line-reference

is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

You may specify EXIT or IOERR but not both.

Description

The OPEN statement makes an external file available for processing by your BASIC 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 causes an exception.

File Attributes: The operating conditions for the file are defined by the file attributes clauses which are as follows:

1. File Access: Specifies what input/output operations are allowed.

Format

```
[,[ACCESS] {INPUT|OUTPUT|OUTIN}]
```

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 and display type files, INPUT is the default if no access is specified. You may not specify INPUT for DISPLAY files when DEVICE PRINTER or DEVICE 3800 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.

   You may not specify OUTIN with files having STREAM organization, or for DISPLAY files where DEVICE PRINTER or DEVICE 3800 is specified.

   For NATIVE type files, OUTIN is the default if no access is specified.

2. File Type: Specifies the appearance of the records in the file.

Format

```
[,[TYPE] {DISPLAY|INTERNAL|NATIVE}]
```

   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. (Each record is a single character string, consisting of the edited values from the output list, positioned according to the specifications.) The string is terminated by an end of record.

      On output, if you specify DEVICE PRINTER in the file-id character string, a carriage control character is prefixed to each record. If you specify DEVICE 3800 in the file-id character 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 the file is to be written as a sequence of numeric and character values. These files are written in internal binary format, each value preceded by a type byte.

   c. NATIVE specifies that the contents of each record are to be formatted by FORM statements. Therefore, if you OPEN a file as native, each READ File or REREAD, WRITE or REWRITE must specify a FORM statement.

      Native is the only file type that can have RELATIVE or KEYED organization.

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 47 on page 308.

	SEQUENTIAL	RELATIVE	KEYED	STREAM
DISPLAY	X			
INTERNAL	X			X
NATIVE	X	X	X	

Figure 47. Allowable Combinations of File Type and File Organization

4. File Pointer: Specifies whether data may be accessed or added at the beginning or end of a file.

Format

```
[,[POINTER] {BEGIN|APPEND|END}]
```

For sequential and stream 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 and for DISPLAY type files opened with OUTIN access.

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 files and for NATIVE and INTERNAL type files opened with OUTIN access.

For relative and keyed files:

a. BEGIN specifies that the file will be opened at its beginning (for OUTPUT files, the only effect is to scratch the file).

b. If the pointer clause is omitted, BEGIN is the default for INPUT and OUTIN files.

c. APPEND/END may not be specified.

d. If the pointer clause is omitted for OUTPUT files, the default is not to position the file and the file is not scratched.

Note: Opening an existing file for any organization with OUTPUT and BEGIN has the same effect as a SCRATCH statement. (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 file. The attributes are:

Format

```
[,[RECORDS] {VARIABLE | FIXED} [rec-length]]
```

a. *Rec-length* is a numeric expression, specifying the actual or maximum length of each record in the file. The length includes carriage and font control characters if DEVICE PRINTER or DEVICE 3800 was specified. The left and right margins are determined from the first character after the carriage and font control characters, if any.

b. Your system administrator has the option of establishing the record type default for nonrelative files as VARIABLE or FIXED at installation. If, for instance, VARIABLE is your system's default value, you need not specify it if you intend to open a file with variable-length records. VARIABLE is the IBM-distributed default for nonrelative files.

   The default for relative files is FIXED. You may not specify VARIABLE for relative files.

c. VARIABLE specifies that records of different lengths may appear in the file. 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 (the maximum may be less for some files; see your IBM BASIC System Services manual).

   If *rec-length* is specified, it must be compatible with the file, or an exception occurs.

   If no *rec-length* is specified and the file already exists or is defined externally to BASIC (see your IBM BASIC System Services manual), the maximum record length defined for the file is used.

   If no *rec-length* is specified and the file does not exist and is not defined externally to BASIC, the maximum record length depends upon the file type:

```
DISPLAY     133
INTERNAL    255
NATIVE      255
```

   These are IBM-distributed default values, which may have been changed by your system administrator during installation.

   If the value of *rec-length* is 0, it is treated as the maximum record length allowed for the file. For the maximum record length, see your IBM BASIC System Services manual. Note that the maximum can never exceed 32756 but for certain files the maximum is less than 32756.

d. 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 file.

   If the *rec-length* is specified and the file already exists, or is defined externally to BASIC (see your IBM BASIC System Services manual), the record length must equal the record length for the file.

   If no *rec-length* is specified and the file already exists or is defined externally to BASIC, the record length defined for the file is used.

   If no *rec-length* is specified, and the file does not exist and is not defined externally to BASIC, the default record length depends upon the file type:

   ```
   DISPLAY     133
   INTERNAL    255
   NATIVE      255
   ```

   These are IBM-distributed default values, which may have been changed by your system administrator during installation.

e. For a VSAM file opened for INPUT, if *rec-length* is specified, that value must be greater than or equal to the maximum defined for the VSAM file; if opened for OUTPUT, the record length must be less than or equal to the maximum defined; if opened for OUTIN, the record length must be equal to the maximum defined.

   In addition, if opened RELATIVE, a specified record length must equal the maximum defined.

   If *rec-length* is not specified, the maximum defined for the VSAM file is used.

For more information, see "BASIC File Capabilities" on page 61.

Exception Conditions: An attempt to open a file that has already been opened results in an exception condition. Attempting to open a file with an invalid file reference number or a file reference number already in use results in an exception condition. These exceptions can be recovered from, if either the IOERR clause is provided, or an EXIT clause refers to an EXIT statement which contains an IOERR clause.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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
```

The OPEN statement in 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 a 3800 printer. The following options are supplied by default (if the IBM-distributed defaults are in effect):

```
SEQUENTIAL, DISPLAY, END, VARIABLE 133.
```

For examples of using keyed and relative files, see your IBM BASIC System Services manual and/or your *IBM BASIC Programming Guide.*

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:

```
ARITHMETIC {DECIMAL|NATIVE}
BASE {0|1}
COLLATE {NATIVE|STANDARD}
INVP [ON|OFF]
{SPREC|LPREC}
PRTZO nn
RD nn
FLAG ({I|W|E|S})
{FIPS|NOFIPS}
```

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 unit or the execution of a program unit. The options FIPS, NOFIPS, FLAG, LPREC and SPREC may also be stated on a COMPILE command, and the options LPREC and SPREC on a RUN command; however, they are overridden by any options explicitly stated in an OPTION statement within a program unit.

If an option is not specified in a program unit and, for FIPS, NOFIPS, FLAG, LPREC and SPREC, is not specified in the COMPILE or RUN command, a default value is used. The IBM-distributed default value is listed for each option below. Check with your system administrator for the initial defaults in effect for your organization. See also "SET OPTION Command" on page 483 for information on changing default values.

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, an error will occur and all declarations except the first are ignored.

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 unit are reactivated when the subprogram is exited.

When a program terminates normally or abnormally inside the BASIC environment, the main 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:

ARITHMETIC { DECIMAL | NATIVE }

This option controls the default typing of identifiers that are not self-typed. These identifiers (if not explicitly typed by a DECIMAL, DEFDBL, DEFINT, DEFSNG, INTEGER, or REAL statement) are typed decimal if the option is DECIMAL or typed real if the option is NATIVE. If one of the typing statements DECIMAL, INTEGER, or REAL is present without a list, the option ARITHMETIC {DECIMAL | NATIVE} will have no effect on the typing of identifiers that are not self-typed.

This option controls the result type of integer division, integer exponentiation, numeric constants, and certain intrinsic functions. The result of integer division and integer exponentiation is decimal if ARITHMETIC DECIMAL is in effect or real if ARITHMETIC NATIVE is in effect (see "Integer Division" on page 52 and "Integer Exponentiation" on page 52 for more information). See "Intrinsic Functions and Array Functions" on page 379 for those functions which are affected by the ARITHMETIC option. See "Numeric Constants" on page 21 for details on typing constants.

This option controls the type of the self-typing character #. # specifies decimal type if ARITHMETIC DECIMAL is in effect. # specifies real double type if ARITHMETIC NATIVE is in effect.

Finally, this option controls the type of DEFSNG and DEFDBL statements. If ARITHMETIC DECIMAL is in effect, the type for both statements is decimal. If ARITHMETIC NATIVE is in effect, the type for DEFSNG is real single and the type for DEFDBL is real double.

If not specified, the default value is used. The IBM-distributed default value is ARITHMETIC DECIMAL.

See "Variables" on page 31 and "Typing" on page 32 for more information on typing. See "Use of Numeric Data Types" on page 26 for advantages of the different data types.

BASE { 0 | 1 }

This option specifies whether subscripts for elements in an array begin at zero or one. This definition applies whether or not a DIM statement was used.

In the absence of a BASE specification, the default value is used. The IBM-distributed default value is BASE 0.

Arrays may be passed as arguments from a program unit to a subprogram which have different bases, but subscripts obey the base of the program unit containing them. Because of the possible confusion different bases could cause, you should 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 COLLATE NATIVE is in effect, the collating sequence is Extended Binary Coded Decimal (EBCDIC).

If COLLATE STANDARD is in effect, the collating sequence is the American National Standard Code for Information Interchange (ASCII).

If neither is specified, the default value is used. The IBM-distributed default value is COLLATE NATIVE.

Character data is always represented in EBCDIC. COLLATE only affects the comparison of character strings (relational expressions, ASORT, DSORT, AIDX, DIDX) and the intrinsic functions CHR$ and ORD. It does *not* affect VSAM keyed files. (Keys are compared using the EBCDIC collating sequence.

(The EBCDIC and ASCII collating sequences are listed in Appendix C, "Character Set Collating Sequences" on page 505.)

INVP [ON | OFF]

INVP (inverted print edit facility) specifies that numeric values are printed interchanging the use of the period (decimal point) and the comma, in order to print in the European format or the U.S. format.

When U.S. format is in effect, a period is printed as the decimal point in a numeric value and a comma is used to separate digits. When European format is in effect, a comma is printed as the decimal point and a period is printed to separate digits.

If you specify INVP with neither ON nor OFF, you reverse the installation default format. INVP ON specifies that numeric values are printed using the European format. INVP OFF specifies that numeric values are printed using the U.S. format.

If the INVP option is not specified, the default value is used. The default value is to use the installation default format.

The IBM-distributed installation default is U.S. format. Note that this installation default is defined for the IBM BASIC Library and therefore depends on the Library used when the program is executed.

This interchange of period and comma applies to all output resulting from the execution of PRINT and PRINT File statements. This includes the commas/periods within an IMAGE statement and within a PIC in a FORM statement.

The following example shows values printed when the IBM-distributed default format is in effect.

Example

```
Without INVP (US)          123,456.78
With INVP (European)       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 controls the precision of real data, and the precision of the PRINT and PRINT File statements.

It specifies the maximum number of significant digits to be printed by the PRINT and PRINT File statements (without the USING clause) when printing numeric values. SPREC specifies "short" precision of 6 digits; LPREC specifies "long" precision of 12 digits.

It also controls the type of the REAL statement when the keyword SINGLE or DOUBLE is not specified. See "REAL Statement" on page 350 for additional details concerning precision.

This option controls the result type of integer division, integer exponentiation, numeric constants, and certain intrinsic functions when ARITHMETIC NATIVE is in effect. The result of integer division and integer exponentiation is real single if SPREC is in effect or real double if LPREC is in effect. See "Numeric Constants" on page 21 for details on typing constants. See "Intrinsic Functions and Array Functions" on page 379 for those functions affected by the ARITHMETIC option.

This option controls the result type of the REAL function. The REAL function returns real single if SPREC is in effect or real double if LPREC is in effect.

If not specified, SPREC or LPREC is as specified for the COMPILE or for the RUN command. And, if not specified for COMPILE or RUN, it is the default value. The IBM-distributed default value is SPREC.

PRTZO nn

This option specifies the width of zones to be used when printing. (See "PRINT without USING Clause" on page 323.)

The value *nn* must be an unsigned integer constant in the range 1 through 255. An exception will occur when the print zone is larger than the difference between the right and left margins (plus 1) at the time a PRINT or PRINT File statement executes.

In the absence of a PRTZO specification, the default value is used. The IBM-distributed default value is PRTZO 20.

RD nn

This option specifies the number of rounded decimal digits to be displayed when a PRINT or PRINT File statement (without the USING clause) 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.

Note: This is the only way you can set or change RD. The default value cannot be changed for this option.

FLAG ({ I | W | E | S })

This option determines the level of error messages reported by the BASIC Processor.

FLAG can also be specified on the COMPILE command (see "COMPILE Command" on page 430).

Control of the error checking level also exists as an option, but FLAG overrides those supplied when the Processor is invoked.

In the absence of a FLAG specification, the default value is used. The IBM-distributed default is FLAG(I).

The levels (in increasing order of severity) are:

I Informative messages
W Warning messages
E Error messages
S Severe error messages

The FLAG option specifies that only errors of levels higher than or equal to the indicated level are to be reported. This option does *not* affect diagnostics produced because of use of the FIPS option.

This option does *not* affect exception messages produced by the BASIC Library. Though not equivalent in function, display of exception messages may be controlled by the ON Condition statement (see "ON Condition Statement" on page 298) and by exception-recovery clauses.

{ FIPS | NOFIPS }

This option indicates whether or not the Processor should produce informational diagnostics for any statement which does not conform to the FIPS BASIC syntax.

FIPS may also be specified on the COMPILE command (see "COMPILE Command" on page 430).

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. The FIPS standard is technically equivalent to the *American National Standard for Minimal BASIC*, ANSI X3.60-1978.

FIPS is not overridden by FLAG{W | E | S}, which directs the system to withhold the display of informational diagnostic messages. (In other words, FIPS informational messages are *not* affected by the FLAG option.)

If neither is specified, the default value is used. The IBM-distributed default value is NOFIPS.

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 BASIC so that it produces the indicated error messages. BASIC remains in this state until:

1. A contradictory immediate OPTION statement or SET OPTION command (see "SET OPTION Command" on page 483) is executed.

2. An INITIALIZE command is executed. (The immediate options are set to the default values.)

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 outside the BASIC environment.

Other Options: All other immediate options (ARITHMETIC, 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 immediate values are set to the default values)

   - The next RUN command is executed (the immediate options are set to the values of the program being executed)

- The next COMPILE command is executed (the immediate values are set to the default values)
- They are altered by an immediate OPTION statement (only affects the options specified in the statement)
- The next INITIALIZE command is executed (the immediate values are set to the default values)

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. The BASE option cannot contradict the base (0 or 1) of any existing arrays (arrays within scope when the OPTION BASE immediate statement is entered). (Scope is defined in "Variables and Arrays in Immediate Statements" on page 134). If you want to change the base, you must DROP existing arrays.

"Immediate Statements" on page 133 gives additional information.

PAUSE Statement

When executing interactively (with a terminal) the PAUSE statement suspends execution of the program in which it appears. The statement is ignored during batch operation.

Format

```
PAUSE [pause-message]
```

where:

pause-message
is an optional character expression.

Description

When the PAUSE statement is processed, program execution is suspended.

To resume execution of the interrupted program, the user enters either a null line or the GO command.

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
```

where *line-number* is the program line where execution stopped.

The following statement, without a specified message, will interrupt the program and display the default message PAUSE AT LINE: 100.

```
100 PAUSE
```

The following statement will interrupt the program and display the message PAUSE HERE TO INSPECT VALUES.

```
150 PAUSE "PAUSE HERE TO INSPECT VALUES"
```

The following example uses a variable for the message PRESS ENTER to be displayed when the program pauses.

```
200 PMSG$ = "PRESS ENTER"
210 PAUSE PMSG$
```

PRINT Statement

The PRINT statement displays data at the terminal.

Format

```
Format 1 (without USING)

    [MAT] PRINT [output-list] [err [,err]...]

Format 2 (with USING)

    [MAT] PRINT USING image-or-form-reference [: [output-list] [err [,err]...]]
```

where:

image-or-form-reference
is the line number or line label of an IMAGE or FORM statement, or a character expression that contains format information identical to that in an IMAGE or FORM statement (for a FORM specification, the keyword FORM must appear).

output-list
is a list of one or more constants, variables, array elements, expressions and/or arrays. List items are separated by commas or semicolons. The keywords TAB (followed by a numeric expression in parentheses), PAGE, and NEWPAGE may also be used in the list of a PRINT without USING.

See "Input/Output Lists" on page 85.

If the *output-list* is omitted, a blank line is printed.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

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 limits of the output line, the print zone, and separators.

Output Line: The output line is first limited to the line size 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. (See "MARGIN Statement" on page 269.)

Print Zone: The print zone is the number of positions allocated for printing one data item: a value sufficiently large to allow printing a floating-point scaled data item in long precision format. A different print zone value may be assigned by the PRTZO option statement. (See "OPTION Statement" on page 312.)

If not specified in an OPTION statement, the print zone value is a system default (the IBM-distributed value is 20). See "SET OPTION Command" on page 483 for how to change the default value.

The print zone is constant throughout a program unit. Items are not limited to the print zone size; 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 it 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. If it appears when the last character position has been filled in a line, an end of line is generated.

Specific uses of the comma and semicolon are explained under each of the three PRINT formats below. They are summarized in Figure 48 on page 322.

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 85 for more information.

In the interactive environment, the PRINT statement allows the user to display data at the terminal during program execution. See your IBM BASIC System Services manual for details of terminal use.

In the batch environment, the result of a PRINT statement is system dependent. See your IBM BASIC System Services manual.

Exception Conditions: All three formats of the PRINT statement may employ the exception-recovery clause to process CONV, SOFLOW, and IOERR conditions.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 48. PRINT Statement—Comma and Semicolon Separator Usage

PRINT without USING Clause

When you don't specify the USING clause 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

You can control the current setting of the margins with the MARGIN statement. See "MARGIN Statement" on page 269.

You can control the width of the print zone with the PRTZO option of the OPTION statement. See "OPTION Statement" on page 312.

Separators: When an embedded or trailing comma appears in the output list, the current line position is advanced to the next print zone.

When an embedded or trailing semicolon appears, the next value appears immediately following the last value, except that a space follows numeric values. This is described in the next section.

If neither a trailing comma nor a semicolon appears, new data is displayed on the next line.

Numeric Data: 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 SPREC is in effect.
- w is 12 if LPREC is in effect.

For additional details of the SPREC | LPREC option, see "OPTION Statement" on page 312.

If a number can be represented exactly as an integer with w or fewer digits, it is displayed using the implicit unscaled representation.

All other numbers are displayed in one of the two explicit forms. If a number can be represented with w or fewer digits in the explicit unscaled form as accurately as it could be in the explicit scaled form, it is displayed in the unscaled form. If not, it is displayed in the scaled form.

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. A printed numeric value is separated from the next value on the line by a space, regardless of the separators in the output list. The space is omitted if it would be at the end of a line.

Character Data: All character values are displayed as is. The current length of a character string determines how many characters will print.

If a numeric or character data item will not fit on the current line, it is printed at the beginning of the next line. Unless, of course, it is the first item on the current line, in which case it is split into line length portions and printed on successive lines.

Example

```
 * LIST
100 DIM A$*20,B$*25,C$*20
110 OPTION PRTZO 25
120 LET A$ = 'HEY DIDDLE DIDDLE, '
130 LET B$ = 'THE CAT AND THE FIDDLE, '
140 LET C$ = 'THE COW JUMPED OVER '
150 PRINT A$, B$, C$
160 PRINT A$; B$; C$
170 PRINT A$; B$; C$;
180 PRINT 'THE MOON'
190 END
 * RUN
HEY DIDDLE DIDDLE,       THE CAT AND THE FIDDLE,  THE COW JUMPED OVER
HEY DIDDLE DIDDLE, THE CAT AND THE FIDDLE, THE COW JUMPED OVER
HEY DIDDLE DIDDLE, THE CAT AND THE FIDDLE, THE COW JUMPED OVER THE MOON
END AT LINE 190.
```

TAB clause: The TAB clause sets the columnar position of the current line. The TAB syntax is:

```
TAB(e)
```

where *e* is a numeric expression. The numeric expression is 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

NEWPAGE or PAGE clause: 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.

Array Data: 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 dimension array prints as a column.

Example

```
100 OPTION BASE 1
110 DIM SAMPLE (6,5)
120   FOR X = 1 TO 6
130     FOR Y = 1 TO 5
140       SAMPLE (X,Y) = X + Y
150     NEXT Y
160   NEXT X
170 MAT PRINT SAMPLE
180 END
 * run
 2                 3                 4                 5
 6

 3                 4                 5                 6
 7

 4                 5                 6                 7
 8

 5                 6                 7                 8
 9

 6                 7                 8                 9
10

 7                 8                 9                 10
11
END AT LINE 180.
```

If line 170 is changed to:

```
170 MAT PRINT SAMPLE;
```

The output looks like:

```
 2  3  4  5  6

 3  4  5  6  7

 4  5  6  7  8

 5  6  7  8  9

 6  7  8  9  10

 7  8  9  10  11
```

PRINT with USING IMAGE Clause

When you use the USING IMAGE clause in the PRINT statement, the printed output is controlled by 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

You can control the settings of the margins with the MARGIN statement. See "MARGIN Statement" on page 269.

Output-List—Scalar Items: Each scalar reference in the output list is edited by the conversion specifications in the IMAGE. If the output list contains at least one item, there must be at least one conversion specification in the IMAGE clause used.

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.

Example

```
 * LIST
100 IMAGE: The quarterback bellowed, ##, ##, ##, HIKE!
110 PRINT USING 100: 32, 65
120 PRINT 'And was immediately sacked because'
130 PRINT 'he forgot the third number'
140 PRINT
150 PRINT USING 100: 87, 53, 21, 75, 33,
160 PRINT 'And passed'
170 PRINT 'for a successful first down'
180 PRINT
190 PRINT USING 100: 55, 34, 89, 90, 22, 79, 41;
200 PRINT 'And lost his job'
210 PRINT 'for overzealous calling'
220 END
 * run
 The quarterback bellowed, 32, 65,
And was immediately sacked because
he forgot the third number

 The quarterback bellowed, 87, 53, 21, HIKE!
 The quarterback bellowed, 75, 33,      And passed
for a successful first down

 The quarterback bellowed, 55, 34, 89, HIKE!
 The quarterback bellowed, 90, 22, 79, HIKE!
 The quarterback bellowed, 41, And lost his job
for overzealous calling
END AT LINE 220.
```

Output-List—Array Items: When an array appears in the output list, the beginning of the array is started on a new 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 left margin of the next line.

If the number of array elements 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 elements 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 you use the USING FORM clause in the PRINT statement, the printed output is controlled by 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

You can control the current settings of the margins with the MARGIN statement. See "MARGIN Statement" on page 269.

At the beginning of the construction of the print record, the entire record contains spaces. Individual data fields are superimposed over these spaces, according to the data-forms and control specifications in the FORM statement. Each item of the output list is matched against the corresponding data-form, converted if necessary to the form and length indicated, and placed in the position designated by the control specification.

If the number of items in the output list is less than or equal to the number of data-form specifications in the FORM statement, any control specifications and literals immediately following the last data-form specification used are also executed.

If the number of items in the output list exceeds the number of data-form specifications, any trailing control specifications and literals are used, and then the FORM is reused from its beginning.

Any valid PAGE or SKIP control (except SKIP 0) causes the output of the current line.

If the output list has been exhausted, the current line is printed 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 (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, but 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 starting at the left margin.

After an output line is displayed, the line position is always reset to the left margin of the next line.

Print-associated FORM statements may designate the control specification X, POS, SKIP, and PAGE, and the data-forms C, G, N, V, and PIC. See "FORM Statement" on page 205.

The following example illustrates how PRINT with USING FORM can be used to generate a complex report that has a standard format with "windows" in which variable values are placed. This format with windows is specified by the FORM statement at line 160. The PRINT statement at line 150 then needs only to refer to the FORM. The PRINT statement uses variables whose values have (in this example) been assigned on lines 100 through 140. In practice, these assignments may have come from a file or some other source.

```
100   LET NAME$     = 'SUSAN ADAMS'
110   LET ADDRESS$  = '123 MAIN STREET'
120   LET SSN%      = 123456789
130   LET AGE%      = 28
140   LET PAY_RATE# = 9.87
150   PRINT USING 160 :NAME$,ADDRESS$,SSN%,AGE%,PAY_RATE#
160   FORM   &
  &   POS12, 'EMPLOYEE SUMMARY REPORT'          ,SKIP3,&
  &   POS10, 'NAME'    ,POS20, C20              ,SKIP, &
  &   POS10, 'ADDRESS',POS20, C20               ,SKIP, &
  &   POS10, 'SSN'     ,POS20, PIC(###/##/####) ,SKIP, &
  &   POS10, 'AGE'     ,POS20, N2               ,SKIP, &
  &   POS10, 'SALARY'  ,POS20, PIC($**.##),' PER HOUR',&
  &                                              SKIP2,&
  &   POS10, '=========================='      ,SKIP3
```

When the program segment above is executed, the windows in the FORM statement are filled with the values of the variables in the PRINT statement. This is how the output will look:

```
EMPLOYEE SUMMARY REPORT


NAME       SUSAN ADAMS
ADDRESS    123 MAIN STREET
SSN        123/45/6789
AGE        28
SALARY     $*9.87 PER HOUR

==========================
```

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 exception-recovery clauses (EXIT, CONV, IOERR, SOFLOW) are not allowed, because such clauses refer to program line numbers or line labels in the workspace.

PRINT FIELDS Statement

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 is zero.

field-definition
is one of the following:

```
character-expression

MAT character-array-name.
```

character-expression *or each* **character-array-name**
must evaluate to:

```
row, column [,[data-form] [,[leading] [,trailing]]]
```

where:

row
is an unsigned, nonzero integer, specifying the row of the display.

column
is an unsigned, nonzero integer, specifying the first column of the display.

data-form
can be one of the data-forms shown in Figure 49 on page 334.

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 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.

output-list
is a list of one or more constants, variables, array elements, expressions and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
CONV line-reference
EXIT line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

Data-form	Meaning
w	Represents either character data or conversion of numeric data to character data, depending upon whether the type of the data is character or numeric.
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.

w
is an unsigned, nonzero integer constant, which may be preceded with spaces.

d
is an unsigned integer constant.

s
is a digit specifier (#, Z, *, $, +, or -), or an insertion character (a comma (,), slash (/), B, or decimal point (.)).

¬
is an exponent specifier, where three or more ¬ characters are shown. The circumflex character (^) and the logical NOT (¬) are equivalent. Use the one your terminal can generate and display.

tr
is a trailing character, (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.

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 49. PRINT FIELDS Statement—Data-form Codes

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 88.)

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 display a message, replace the value with zero, and continue executing.

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 unsigned, 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 (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 as 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. See "FORM Statement" on page 205.

If the data-form definition is *W*, the data is displayed as follows:

- Character strings are left-justified in the field with nulls padded on the right. If the string length is greater than the field length, a string overflow (SOFLOW) exception occurs.

- Numeric values are displayed according to the rules for the data-form specified as shown in Figure 49 on page 334. 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 containing an attribute 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.

If no attribute is specified, the attribute of the previous field (if any) carries over. If there was no preceding attribute specified for this page, 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.

For more information, see "Input/Output Data Rules" on page 86.

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, and so forth).
- Some stimulus external to BASIC causes the screen display to change (for example, the CLEAR key is pressed).

Exception 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 statement, the IOERR exception occurs.

These exceptions can be recovered from, if the CONV, IOERR, or SOFLOW clause is specified, or if an EXIT clause refers to an EXIT statement that contains these clauses.

The I/O exception conditions interact with the ON Condition statement as described in "Exception Handling in I/O Statements" on page 128.

Example

```
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 (through column 26) is filled with spaces.

Example

```
100 A$ = "22,1,C 3"
110 PRINT FIELDS A$: "AGE"
```

Prints AGE at location row 22, beginning at column 1.

Example

```
130 DATA "10,20,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" on page 248 for the program in which this statement is used.

PRINT File Statement

The PRINT File statement transmits data to a display type file.

Format

```
[MAT] PRINT #fileref [[,] USING image-or-form-reference]
    [[,] FONT expression] [: [output-list] [err [,err]...]]
```

where:

fileref
is a numeric expression which, when evaluated and rounded, must be an unsigned integer within the range 0 to 255 which identifies the file to be processed.

image-or-form-reference
is the line number or line label of an IMAGE or FORM statement, or a character expression which contains format information identical to that in an IMAGE or FORM statement (for a FORM specification, the keyword FORM must appear).

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 one or more constants, variables, array elements, expressions and/or arrays. List items are separated by commas or semicolons. The keywords TAB (followed by a numeric expression in parentheses), PAGE, and NEWPAGE may also be used in the list of a PRINT File without USING.

See "Input/Output Lists" on page 85.

If the *output-list* is omitted, a blank line is written.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
ENDPAGE line-reference
EOF line-reference
IOERR line-reference
PAGEOFLOW line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

Description

The PRINT File statement must refer to a file having display type.

For more information on files, see "BASIC File Capabilities" on page 61. 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 85 for more information.

Fileref: The fileref must refer to a display type file. (See "Combinations of File Type, Organization, Access, and Records" on page 64.) A PRINT File statement with a fileref of 0 is equivalent to a PRINT statement (see "PRINT Statement" on page 320 and "OPEN Statement" on page 305).

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 320.

FONT Clause: The FONT clause is used in conjunction with the DEVICE 3800 clause in the file-id of the OPEN statement to specify which font on the 3800 printer is to be used. If not specified, the default is FONT 1.

The expression following FONT must be a numeric expression. The rounded integer value of the expression must be between 1 and 4 inclusive. The specified or default FONT may be different than the FONT used in a preceding PRINT File that ended with a semicolon or comma (thereby allowing the FONT to change in the middle of an output line). For more information on fonts, see the *IBM 3800 Printing Subsystem Programmer's Guide.*

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.

For more information, see "Input/Output Data Rules" on page 86.

Use of the TAB, PAGE, or NEWPAGE clause is explained in "PRINT without USING Clause" on page 323.

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 305), an exception is generated. The SYSTEM action for this exception is a warning message.

Exception Conditions: The 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 320.

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 exception-recovery clause.

The ENDPAGE (or PAGEOFLOW) condition occurs if the PRINT File attempts to start a new line beyond the bottom margin. (See "MARGIN File Statement" on page 273.)

For a PRINT File statement with a file reference number of 0, the EOF condition has no meaning. If specified, it is ignored.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

Example

```
10 PRINT #4, FONT 2: A,B,
20 PRINT #4, FONT 1: C,D
```

will print the values of A, B, C, and D on the same line in zones with the values of A and B printed in FONT 2 and the values of C and D printed in FONT 1.

Note: Changing the FONT as done above causes two output lines to be generated (the second overprints the first).

PUT Statement

The PUT 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 an unsigned integer within the range 1 to 255, and which identifies the file to be processed.

output-list
is a list of one or more constants, variables, array elements, expressions and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
EOF line-reference
IOERR line-reference
```

line-reference
is a line number or line label.

The EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

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 binary format, and are preceded by an identifying byte. Values from arrays are written with the rightmost subscripts varying most rapidly.

For more information on files, see "BASIC File Capabilities" on page 61.

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 85 for more information.

Output-List: See "Input/Output Data Rules" on page 86.

Fileref: The fileref must refer to a stream file. (See "Combinations of File Type, Organization, Access, and Records" on page 64.)

Exception Conditions: If values remain to be written on the file, but space for the file is exhausted, an end-of-file (EOF) 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 file reference of 0.

Both EOF and IOERR conditions may be recoverable if the corresponding exception recovery clause is used in the PUT statement or in a referenced EXIT statement.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 character 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

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 RND intrinsic 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 (unpredictable). See "RND[(X)] Random" on page 407 for additional information on random number generation.

Immediate Execution

The immediate RANDOMIZE statement operates with the same restrictions and capabilities as the non-immediate RANDOMIZE statement, as described above.

READ Statement

The READ statement retrieves the internal data table created by DATA statements.

Format

```
[MAT] READ input-list [err [,err]...]
```

where:

input-list
is a list of one or more variables, array elements, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
EOF line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

Description

When a READ statement is executed, successive values are assigned from the internal data table to the items in the input list. If all of the values of the data table 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 table or to a specific position within the data table.

See also "DATA Statement" on page 174, and "RESTORE Statement" on page 358.

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 85 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 (SOFLOW) 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 (OFLOW) occurs. 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 table with the rightmost subscript varying most rapidly, starting at the current data table 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.

For more information, see "Input/Output Data Rules" on page 86.

Exception Conditions: Conversion, string overflow, and end of internal data table exceptions may be recovered from if the appropriate exception-recovery clause is included in the READ statement or a referenced EXIT statement. (IOERR may be used instead of EOF in an EXIT statement referenced by a READ statement to recover from end of internal data table exception. If both are specified, the line-reference for the EOF clause is used.)

Numeric overflow and underflow can be handled by the ON Condition statement.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

Example

```
100 OPTION BASE 1
200 DATA ABCD,123.4,2,4,"XYZ",7*"ZZZ"
300 READ A$,B
400 DIM Z$(10,10)*3
500 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 elements and assigned the remaining values XYZ, and 7 repetitions of ZZZ, in the data table.

READ File Statement

The READ File statement retrieves a record from a native or an internal sequential file.

Format

Format 1 (native files)

```
[MAT] READ #fileref [,] USING form-reference [[,]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 an unsigned integer within the range 1 to 255 and which identifies the file to be processed.

form-reference
is the line number or line label of a FORM statement, or a character expression that contains format information identical to that in a FORM statement (the keyword FORM must appear).

pos
is one of the following:

```
KEY [rel] character-expression
SEARCH [rel] character-expression
REC[ORD] [=|EQ] numeric-expression
```

where:

rel
is one of the following:

```
{ = | => | >= | EQ | GE }
```

If *rel* is omitted, = is assumed.

Note: The *#fileref*, USING, and *pos* clauses can occur in any sequence.

input-list
is a list of one or more variables, array elements, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err

is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
EOF line-reference
IOERR line-reference
NOKEY line-reference
NOREC line-reference
SOFLOW line-reference
```

line-reference

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 exception-recovery clauses are mutually exclusive.

Description

The execution of a READ File 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

Fileref: For a NATIVE file type, the organization of the file may be sequential, relative, or keyed. For an INTERNAL type file, the organization must be sequential.

The READ File statement may *not* be used with a DISPLAY type file or with a STREAM organization file.

For both native files and internal files, the next sequential record is retrieved by the READ File statement with no KEY, SEARCH, or RECORD clause; for relative files, the next sequential record is the next nonnull 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.

After 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.

For more information on files, see "BASIC File Capabilities" on page 61.

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 85 for more information.

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.

Example

```
100 READ #1 USING "FORM C20":C$
```

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 elements may not be exceeded.

For more information, see "Input/Output Data Rules" on page 86.

Exception Conditions: Various exception 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 exception (SOFLOW) 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, opened as sequential, if the items of the input list are all used and more data remains in the record, a conversion exception (CONV) 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, opened as sequential, a conversion exception (CONV) occurs if there is not enough data in the record to fill all the input items.

For native files, a conversion exception (CONV) 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 end-of-file (EOF) exception occurs when no KEY, SEARCH, or RECORD is specified and the last record of the file has already been read.

The NOREC conditon occurs if no relative record satisfies the RECORD condition for a relative file.

The NOKEY condition occurs if no record satisfies the KEY or SEARCH condition for a keyed file.

The IOERR exception occurs if the file was opened with DISPLAY type or with STREAM organization.

The IOERR exception also occurs if a hardware malfunction occurs or some other error occurs which prevents the reading of the record.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

Example

```
 10 TWO% = 2
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 File assigns values from the next sequential record of an internal file type to two variables, and ignores the rest of the record. The second READ File 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 File 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.

REAL Statement

The REAL statement specifies which identifiers are to be assigned real type.

Format

```
REAL [SINGLE|DOUBLE] [{identifier|(letter-list)} [,{identifier|(letter-list)}]...]
```

where:

identifier
is a 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.

Description

The REAL statement declares a specific identifier or any identifier beginning with a specific letter as having real type.

The precision is determined as follows:

- Single, if SINGLE is specified.
- Double, if DOUBLE is specified.
- Single, if neither SINGLE nor DOUBLE is specified and the SPREC option is in effect.
- Double, if neither SINGLE or DOUBLE is specified, and the LPREC option is in effect.

A real double identifier may end with the self-typing character # if the ARITHMETIC NATIVE option is in effect. A real identifier may not end with the self-typing character % or $. A real single identifier may not end with the self-typing character #.

A *letter-list* within parentheses indicates that all identifiers beginning with those letters are to be typed REAL with the precision as determined above, unless they end in a contradictory self-typing character (#, %, or $), or unless they are explicitly declared otherwise in a DECIMAL, DEFDBL, DEFINT, DEFSNG, INTEGER, or REAL statement.

If a REAL statement specifies no identifiers and no letter list, the default type for all identifiers in the program unit is set to real with the precision as determined above. This is also the default if the ARITHMETIC NATIVE option is in effect (see "OPTION Statement" on page 312).

REAL statements may appear anywhere in a program unit and affect identifiers throughout that unit. The identifiers affected may be variable names, array names, or user-defined function names.

Example

```
100 REAL ALPHA,(R-T,E,G-H),XI
```

specifies that identifiers ALPHA and XI, as well as all identifiers beginning with the letters R, S, T, E, G, and H are typed real. If the program unit subsequently contains a variable named EPSILON, it would be assigned real type. However, GAMMA# would be typed decimal (if ARITHMETIC DECIMAL is in effect), since the # overrides the first letter typing.

See "Variables" on page 31 and "Arrays" on page 36 for more information.

Example

```
100 REAL SINGLE SIGMA, NU
```

and

```
100 OPTION SPREC
110 REAL SIGMA, NU
```

both specify the same typing for SIGMA and NU; identifiers SIGMA and NU are typed real single.

Immediate Execution

You can use the REAL statement to set the type of immediate variables and arrays. The format and description are the same as for a REAL statement in a program.

See "Immediate Statements" on page 133 and "Immediate Type and Dimensions" on page 135 for the rules regarding the interaction with other immediate statements and program statements.

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, such as GOTO. Execution then continues with the next line after the REM statement.

Because 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).

See also "Comments" on page 15 for additional information on REM and exclamation mark comments.

Comments Using the Exclamation Mark

Remarks may also be appended to statements by using an exclamation mark. This is sometimes called a trailing comment.

The exclamation mark comment (!), like the REM remark, 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 or DATA statement line because the exclamation mark can have meaning within those statements.

Example

```
100 REM TEST NUMBER
110 IF NUMBER LT 200 THEN GO TO 200
```

is functionally equivalent to

```
100 IF NUMBER LT 200 THEN GOTO 200   !TEST NUMBER
```

As with the REM statement, an exclamation mark comment must be the last entry on a line. Unlike REM statements, a statement with an exclamation mark comment can be continued; an ampersand as the last nonblank character on the line indicates continuation. However, it is *not* the exclamation mark comment itself that is continued; it is the statement before the comment that is continued.

Note: A statement beginning with an exclamation mark is treated the same as a REM statement and not as an exclamation mark comment. See also "Comments" on page 15.

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
```

Immediate Execution

A REM statement can be used to enter comments. REM statements can be useful for placing comments in the log file (see "SET LOG Command" on page 479) or in a profile (see "PROFILE Command" on page 462).

An immediate REM statement may be specified using the REM keyword or an exclamation mark (!).

REREAD Statement

The REREAD statement makes the last accessed record in a native file available again.

Format

```
[MAT] REREAD #fileref [,] USING form-reference: input-list [err [,err ]...]
```

where:

fileref
is a numeric expression which, when evaluated and rounded, is an unsigned integer within the range 1 to 255, and which identifies the file to be processed.

form-reference
is the line number or line label of a FORM statement, or a character expression that contains format information identical to that in a FORM statement. The keyword FORM must appear if you use the character expression in the form-reference.

Note: The *#fileref* and *USING* clauses may occur in any order.

input-list
is a list of one or more variables, array elements, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
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 exception-recovery 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 File statement or another REREAD statement.

For more information on files, see "BASIC File Capabilities" on page 61.

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 85 for more information.

Fileref: The fileref must refer to a native file. (See "Combinations of File Type, Organization, Access, and Records" on page 64.)

USING Clause: The data in the record is formatted according to the specifications of the FORM statement.

The USING clause can itself contain a character expression that specifies a format for the data.

Example

```
100 REREAD #1 USING "FORM C20":C$
```

Input-List: The data in the record is assigned to the variables in the input list, in the same manner as the READ File statement (see "READ File Statement" on page 346).

For more information, see "Input/Output Data Rules" on page 86.

Exception Conditions: The exception conditions IOERR, CONV, and SOFLOW may be recoverable if they are specified on the statement or in a referenced EXIT statement.

A CONV condition 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 exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 File 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 a file pointer.

Format

```
RESET #fileref [[,]pos] [: [err [,err]...]]
```

where:

fileref
is a numeric expression which, when evaluated and rounded, must be an unsigned integer within the range 1 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
```

where:

rel
is one of the following:

```
{= | >= | => | EQ |GE}
```

If *rel* is omitted, = is assumed.

Note: The *#fileref* and *pos* clauses may be in any sequence.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
IOERR line-reference
NOKEY line-reference
NOREC line-reference
```

line-reference
is a line number or line label.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

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.

Description

Any type of external file may be repositioned with a RESET statement.

Pos Clause: The BEGIN clause positions any file to its beginning; if no positioning clause is specified, BEGIN is assumed. Resetting an OUTPUT file to its beginning does not scratch the file. To output more records starting at the beginning, the file must be empty (see "SCRATCH Statement" on page 365).

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.

See also "BASIC File Capabilities" on page 61.

Exception Conditions: The conditions IOERR, NOREC, and NOKEY may be recovered if they are included in exception-recovery clauses or a referenced EXIT statement (see "EXIT Statement" on page 197).

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 exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 lets you read the same data items more than once by restoring the data table pointer to the beginning, or other specified line, of the internal data table.

Format

```
RESTORE [data-reference]
```

where:

data-reference
is the line number or line label of a DATA statement in the program unit.

Description

The RESTORE statement sets or resets the pointer of the internal data table, created by DATA statements and accessed by READ statements.

If you do not specify a *data-reference*, the RESTORE statement resets the pointer to the first value in the internal data table.

When you specify a *data-reference*, the pointer is set to the first data item in the specified DATA statement whether or not that item has been previously read.

A RESTORE statement without a *data-reference* is ignored if there are no DATA statements in the program unit.

A DATA statement must be the first and only statement on a line if it is referenced in a RESTORE statement. (A DATA statement does not have to be the first statement on a line if it is not referenced in a RESTORE statement.)

You will get an error message if your *data-reference* refers to a nonexistent line, a line that does not contain a DATA statement, a line where the DATA statement is not the first and only statement on the line, or a line that is not in the program unit.

See also "DATA Statement" on page 174, "READ Statement" on page 344, and "Data Table Statements" on page 87.

See "RESET Statement" on page 356 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 internal data table was repositioned to its first value.

Example

```
100 DATA 1,2,3
200 PRINT "START"
300 DATA 4,5,6
400 READ U,V
500 RESTORE
600 READ A,B,C,D,E
700 RESTORE 300
800 READ X,Y,Z
```

In this example, line 400 assigns the values 1 and 2 to variables U and V. Line 500 resets the data table pointer to the first value (1) of the first DATA statement (line 100). Line 600 then assigns values 1, 2, and 3 from the first DATA statement as well as values 4 and 5 from the second DATA statement (line 300) to A,B,C,D, and E. Line 700 resets the data table pointer to the first value of the second DATA statement. Line 800 assigns the values 4, 5 and 6 to the variables X, Y, and Z.

Example

```
100 DATA 123,456,789
200 MORE: DATA ABC
300 READ A, B        !Assign 123 to A and 456 to B.
400 RESTORE          !We need 123 again.
500 READ C
600 RESTORE MORE     !Skip 456 and 789.  We need DATA ABC.
700 READ X$
```

In this example, the value assigned to C is 123, because the RESTORE in line 400 repositions the data table pointer to the first value in the first DATA statement (line 100). X$ will contain value ABC because the RESTORE in line 600 sets the pointer to the values specified in the line labeled MORE (line 200).

RETRY Statement

The RETRY statement reprocesses a statement which caused an exception.

Format

```
RETRY
```

Description

Execution of a RETRY statement results in reprocessing the statement that 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.

See also "CONTINUE Statement" on page 173, "ON Condition Statement" on page 298, and "Exception Handling Statements" on page 127.

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 or ON GOSUB statement executed, completing a GOSUB/RETURN cycle. See "GOSUB Statement" on page 228, and "ON GOTO/GOSUB Statement" on page 303.

Execution of a RETURN statement returns the last active GOSUB or ON GOSUB statement to inactive status.

The execution of a RETURN statement without an active GOSUB or ON GOSUB statement results in an exception.

It is not necessary that equal numbers of GOSUB and ON GOSUB statements and RETURN statements be executed before termination of a program unit or multi-statement function (execution of a SUBEXIT or END SUB statement in a subprogram or of a FNEND statement in a function). All active GOSUB and ON 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 73.

REWRITE Statement

A REWRITE statement updates an existing record stored in a native file.

Format

```
[MAT] REWRITE #fileref [,] USING form-reference
    [[,] pos] : output-list [err [,err]...]
```

where:

fileref
is a numeric expression which, when evaluated and rounded, must be an unsigned integer within the range 1 to 255, and which identifies the file to processed.

form-reference
is the line number or line label of a FORM statement, or a character expression that contains format information identical to that in a FORM statement.

pos
is one of the following:

```
KEY [=|EQ] character expression

REC[ORD] [=|EQ] numeric expression
```

Note: The *#fileref*, USING, and *pos* clauses may occur in any sequence.

output-list
is a list of one or more constants, variables, array elements, expressions, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
EOF line-reference
IOERR line-reference
NOKEY line-reference
NOREC line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. The same exception-recovery clause may appear only once.

Description

The REWRITE statement updates an existing record of a native file which has been opened with the access attribute OUTIN.

For more information on files, see "BASIC File Capabilities" on page 61.

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 85 for more information.

Fileref: The fileref must refer to a native file opened with access OUTIN. (See "Combinations of File Type, Organization, Access, and Records" on page 64.)

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.

Example

```
100 REWRITE #1 USING "FORM C20":C$
```

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. Note that portions of the record not rewritten will be preserved (the record will not be shortened).

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.

For more information, see "Input/Output Data Rules" on page 86.

Exception Conditions: The conditions IOERR, NOREC, and NOKEY may be recovered if they are included in exception-recovery clauses or a referenced EXIT statement (see "EXIT Statement" on page 197).

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.

A CONV condition occurs if a value cannot be converted as specified in the FORM.

The CONV condition occurs if the record generated exceeds the record length defined for the file. Note that all the values for the output list must be placed in a single record.

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 (keyed files only).

The IOERR condition occurs if an attempt is made to lengthen an existing record when writing to a sequential file. Note that the CONV condition will occur instead if this length is also longer than allowed for the file.

The IOERR condition also indicates that a hardware malfunction or other cause prevents the writing of the altered record.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 an integer within the range 1 to 255, and which identifies the file to be scratched.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
IOERR line-reference
```

line-reference
is a line number or line label.

You may specify either EXIT or IOERR, but not both.

An EXIT clause must refer to the line number or line label of an EXIT statement.

Description

Processing the SCRATCH statement erases all the values or records in a file and resets the file pointer to the beginning of the file.

The file must be opened for OUTPUT or OUTIN access (see "OPEN Statement" on page 305).

For more information on files, see "BASIC File Capabilities" on page 61.

Exception Conditions: An IOERR condition occurs if a hardware malfunction or other condition prevents the file from being scratched. If an IOERR exception-recovery clause is specified in the SCRATCH statement or on a referenced EXIT statement, the IOERR condition may be recoverable.

Example

```
150 SCRATCH #1: IOERR 900
```

deletes the contents of the file opened as #1. If the file hasn't been opened, control will pass to line 900.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 79.

The SELECT block consists of four parts, coded in the following order:

1. The SELECT statement containing the selection expression.
2. Any number of CASE blocks. A CASE block is defined as the statements between one CASE statement and another, between a CASE statement and a CASE ELSE statement, or between a CASE statement and an END SELECT statement.
3. An optional CASE ELSE block, which includes all statements between the CASE ELSE statement and the END SELECT statement.
4. An END SELECT statement 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.

See also "CASE Statement" on page 162, "CASE ELSE Statement" on page 164, and "END SELECT Statement" on page 195.

An example of a SELECT block is given in "SELECT Blocks" on page 79.

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 (see "END Statement" on page 193). 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 rounded integer value of the numeric expression is returned to the operating system. If the numeric expression is omitted, zero is returned.

Inside the BASIC environment, the rounded integer value of the numeric expression (if nonzero), is displayed as part of the ending message.

Example

```
110 STOP
```

The above statement, without a numeric expression, will stop the program and set the return code to 0 (the default value).

Example

```
110 RETCODE = 139
120 STOP RETCODE
```

This example, with a numeric expression, will stop the program and set the return code to 139.

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 (that is, the resulting return code) is allowed, with the restriction it does not refer to function definitions, but it is ignored.

SUB Statement

The SUB statement begins and names a subprogram.

Format

```
SUB name [(parameter [,parameter]...)]
```

where:

name
is a subprogram identifier of 1 to 7 characters. *name* can *not* be one of the names interpreted as a CALL to the interface routines for other language program units:

```
BASIC, CLINK, COBOL, FLINK, FORTRAN,
GDDM, PLI, PLINK, SQL, or SYSTEM
```

parameter
may be either a simple variable or an empty array declarator.

where:

empty array declarator
has the form:

```
identifier ([,]...)
```

There may be no more than six commas. However, the number of commas is not checked against the number of dimensions of the argument and has no effect.

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.

The SUB statement must be the first statement on a line.

Parameters in a SUB statement may be typed in DECIMAL, DEFDBL, DEFINT, DEFSNG, INTEGER, or REAL statements which follow in the program unit.

The number, type, and precision of arguments in a CALL statement must agree with the number, type, and precision of parameters in the corresponding SUB statement. An array used as an argument must correspond to a parameter which is an empty array declarator.

An array that is a parameter (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 COMMON 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 arguments 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 arguments are given in "CALL Statement" on page 149.

Example

```
100 SUB RTN(A(,))
```

In this example, the SUB statement names the subprogram as RTN, and also declares array A to have two dimensions; however, the values of the dimensions are supplied by the arguments 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 BASIC Subprograms" on page 95, "SUBEXIT Statement" on page 370, and "END SUB Statement" on page 196.

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 statement following the CALL statement.

See "SUB Statement" on page 368, "CALL Statement" on page 149, "Subprogram Statements" on page 93 and "END SUB Statement" on page 196.

TRACE Statement

When debugging is active, the TRACE statement turns tracing ON or OFF.

Format

```
TRACE ON [TO #fileref]
    or
TRACE OFF
```

where:

fileref
is a numeric expression which, when evaluated and rounded, must be an integer within the range 0 to 255, and which identifies the file for the trace listing.

Description

The TRACE statement displays the flow of control within a program.

When debugging is active (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.

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). A nonzero fileref must designate a file that was opened as OUTPUT, DISPLAY, and SEQUENTIAL.

If debugging was activated by a DEBUG ON TO *fileref* statement, the TO *fileref* clause on a TRACE ON statement is ignored and the *fileref* on the DEBUG statement is used.

A DEBUG 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 file reference 0.

TRACE statements are ignored in compiled program units.

See also "Debugging Statements" on page 130 and "DEBUG Statement" on page 176.

Immediate 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. (Only the first action described above is done.)

Note that an immediate TRACE statement will have no effect on compiled program units.

USE Statement

The USE statement establishes a name correspondence in a chained program for those names passed via a CHAIN statement.

Format

```
USE parameter [,parameter]...
```

where:

parameter
is a simple variable or array name.

Note that array names in a USE statement must not be followed by an empty array declarator as done for a parameter in a SUB statement.

Description

Prior to execution of the chained main program, the attributes passed for each variable or array in the USE statement are matched against the defined attributes for the same variable or array in the chaining program unit. The matching local variables and arrays in the chained main 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 a name matches, but the type, maximum number of elements for an array, or the maximum character length for a character variable or array do not match, then an exception occurs which will cause the program to terminate. (Passed attributes are not automatically inherited by the chained program. An array name in the USE statement must be defined by some other statement, 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 main chaining program unit. The data from VALUEA is passed to PROGB, the chained main 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 unit.

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 parameters in the USE statement define the data items retained by the main program. All other data values (except COMMON variables) are set to zero or null prior to their first use.

A USE statement may *not* appear in a subprogram.

See "CHAIN Statement" on page 166 and "Chaining Statements" on page 96.

WRITE Statement

The WRITE statement adds records to native and internal files.

Format

Format 1 (native type)

```
[MAT] WRITE #fileref [,] USING form-reference [[,] pos] : output-list [err [,err]...]
```

Format 2 (internal type)

```
[MAT] WRITE #fileref : output-list [err [,err]...]
```

where:

fileref
is a numeric expression which, when evaluated and rounded, must be an unsigned integer within the range 1 to 255, and which identifies the file to be processed.

form-reference
is the line number or line label of a FORM statement, or a character expression that contains format information identical to that in a FORM statement. The keyword FORM must appear if you use the character expression in the form-reference.

pos
is

```
REC[ORD] [= | EQ] numeric expression
```

Note: The *#fileref*, USING, and *pos* clauses may appear in any order.

output-list
is a list of one or more constants, variables, array elements, expressions, and/or arrays. List items are separated by commas.

See "Input/Output Lists" on page 85.

err
is one of the following exception-recovery clauses:

```
EXIT line-reference
CONV line-reference
DUPKEY line-reference
DUPREC line-reference
EOF line-reference
IOERR line-reference
SOFLOW line-reference
```

line-reference
is a line number or line label.

An EXIT clause must refer to the line number or line label of an EXIT statement.

If an EXIT clause is specified, none of the other exception-recovery clauses may be specified. For example, you may specify either EXIT or IOERR, but not both. The same exception-recovery clause may appear only once.

Description

The WRITE statement adds a new record to native and internal files. If the record already exists in a native file, the REWRITE statement should be used to rewrite the record.

For a native type file, the organization of the file may be sequential, relative, or keyed:

Sequential The record may only be added to the end of the file.

Relative The record may be added anywhere in the file as long as the record does not already exist. The *pos* clause is required.

Keyed The record may be added anywhere in the file as long as the record does not already exist.

For an internal type file, the organization of the file may be sequential or stream. For both organizations, records may only be added to the end of the file.

Note: Opening an existing file with OUTPUT and BEGIN has the same effect as a SCRATCH statement; all data in the file is lost and the end of the file coincides with the beginning. Opening an existing file with OUTIN and BEGIN does not cause existing records to be scratched.

For more information on files, see "BASIC File Capabilities" on page 61.

For a NATIVE file or an INTERNAL SEQUENTIAL file, a WRITE statement generates a single record (all data in the output list must fit within the record).

For a STREAM file, format 2 of the WRITE statement is equivalent to a PUT (each data item in the output list is written to a separate record).

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 85 for more information.

Output-list: See "Input/Output Data Rules" on page 86.

Fileref: The fileref must refer to a native or internal file, opened for OUTPUT or OUTIN access. (See "Combinations of File Type, Organization, Access, and Records" on page 64.)

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 spaces.

For internal and stream files, the USING clause is invalid; the values are written to the file in internal binary format.

The USING clause can itself contain a character expression that specifies a format for the data.

Example

```
100 WRITE #1 USING "FORM C20":C$
```

Pos Clause: A WRITE statement with a RECORD clause is used to add a record to a relative file. The rounded value of the numeric expression designates the record number. The RECORD clause must be specified if and only if the file was opened with RELATIVE organization. (See "Relative Organization" on page 63 for more information on record numbers.)

For a keyed file, a record is added by a WRITE statement with the key formatted in the record at the position and with the length established for the file.

For a sequential or stream file, the file must be positioned at the end, and the record or records will be added at the end.

Exception Conditions: Several conditions may be recoverable if the appropriate exception-recovery clause is included in the WRITE statement or on a referenced EXIT statement (see "EXIT Statement" on page 197).

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 CONV condition occurs if the record generated exceeds the record length defined for the file. Note that all the values for the output list must be placed in a single record when writing to a NATIVE file or to an INTERNAL SEQUENTIAL file; when writing to a STREAM file, each value is placed in its own record.

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 occurs if a WRITE statement does not have a *pos* clause (RECORD) when writing to a RELATIVE file.

The IOERR condition is also caused by hardware malfunction or other conditions which prevent the writing of the record, such as attempting to write an existing record in a sequential file.

The exception-recovery clauses interact with the ON condition statement as described in "Exception Handling in I/O Statements" on page 128.

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 spaces. If an end-of-file occurs, control would pass to line 600. An IOERR exception would result in control being transferred to line 500.

Intrinsic Functions and Array Functions

Notation Used for Parameters of Intrinsic Functions

In this section, those intrinsic functions that require arguments are indicated by a parameter list that is enclosed in parentheses and is placed after the function name. More than one parameter identifier is given for each parameter type in the legend below. This permits the use of different identifiers where more than one argument of the same type is required by a function. The notation used for parameters is:

Notation	Meaning
X, Y, or Z	Numeric parameter which can be integer, decimal, or real. `Example: ABS(X)`
M	Integer parameter. The function can be used with any numeric expression as the argument, but if the result of the expression is decimal or real, it will automatically be converted to integer, with rounding (that is, the value IFIX(M) is used). `Example: FILE(M)`
A or B	Numeric array parameter. `Example: DET(A)`
C$, D$, or E$	Character parameter. `Example: LEN(C$)`
A$	Character array parameter. `Example: ARY(A$)`

List of Intrinsic Functions and Array Functions

Function Name	Purpose	Category
ABS(X)	Absolute Value	Intrinsic
ACOS(X)	Arccosine	Intrinsic
AIDX(A) or AIDX(A$)	Ascending Index	Array
ANGLE(X,Y)	Angle	Intrinsic
ASIN(X)	Arcsine	Intrinsic
ASORT(A) or ASORT(A$)	Ascending Sort	Array
ATN(X)	Arctangent	Intrinsic
CEIL(X)	Ceiling	Intrinsic
CEN(X)	Fahrenheit to Centigrade	Intrinsic
CHR$(M)	Character	Intrinsic
CNT	Count	Intrinsic
CODE	Code	Intrinsic
CON	Constant	Array
COS(X)	Cosine	Intrinsic
COSH(X)	Hyperbolic Cosine	Intrinsic
COT(X)	Cotangent	Intrinsic
CSC(X)	Cosecant	Intrinsic
DAT$[M]	(Year/Month/Day)	Intrinsic
DATE	(Year, Number of Days)	Intrinsic
DATE$	(Year/Month/Day)	Intrinsic
DBL(X)	Real Double	Intrinsic
DEC(X)	Decimal	Intrinsic
DEG(X)	Radians to Degrees	Intrinsic
DET[(A)]	Determinant	Intrinsic
DIDX(A) or DIDX(A$)	Descending Index	Array
DOT(A,B)	Dot Product	Intrinsic
DSORT(A) or DSORT(A$)	Descending Sort	Array
EPS	ε	Intrinsic
ERR	Exception Code	Intrinsic
EXP(X)	Exponential Value	Intrinsic
FAH(X)	Centigrade to Fahrenheit	Intrinsic
FILE(M)	File Status	Intrinsic
FILENUM	File Number	Intrinsic
FILE$(M)	File Name	Intrinsic
FP(X)	Fractional Part	Intrinsic
IDN	Identity	Array
IFIX(X)	Rounded Integer Value	Intrinsic
INF	Infinity	Intrinsic
INT(X)	Largest Integer	Intrinsic
INV(A)	Inverse	Array
IP(X)	Integer Part	Intrinsic
JDY[(C$)]	Julian Date	Intrinsic
KEYNUM	Key Number	Intrinsic
KLN(M)	Key Length	Intrinsic

Function Name	Purpose	Category
KPS(M)	Key Position	Intrinsic
LEN(C$)	Length	Intrinsic
LINE	Line Number	Intrinsic
LOG(X)	Natural Logarithm	Intrinsic
LOG2(X)	Base 2 Logarithm	Intrinsic
LOG10(X)	Common Logarithm	Intrinsic
LPAD$(C$,M)	Left Pad	Intrinsic
LTRM$(C$)	Left Trim	Intrinsic
LWRC$(C$)	Lower Case	Intrinsic
MAX(X,Y[,Z]...)	Maximum	Intrinsic
MIN(X,Y[,Z]...)	Minimum	Intrinsic
MOD(X,Y)	Modulo	Intrinsic
NUL$	Null String	Array
ORD(C$)	Ordinal Position	Intrinsic
PARM$	Parameter	Intrinsic
PI	π	Intrinsic
POS(C$,D$)	Position	Intrinsic
POS(C$,D$,M)	Position	Intrinsic
PRD(A)	Array Product	Intrinsic
RAD(X)	Degrees to Radians	Intrinsic
REAL(X)	Real	Intrinsic
REC(M)	Record Number	Intrinsic
REM(X,Y)	Remainder	Intrinsic
RLN(M)	Record Length	Intrinsic
RND[(X)]	Random	Intrinsic
ROUND(X,M)	Round	Intrinsic
RPAD$(C$,M)	Right Pad	Intrinsic
RPT$(C$,M)	Repeat	Intrinsic
RTRM$(C$)	Right Trim	Intrinsic
SEC(X)	Secant	Intrinsic
SGN(X)	Sign	Intrinsic
SIN(X)	Sine	Intrinsic
SINH(X)	Hyperbolic Sine	Intrinsic
SIZE(A) or SIZE(A$)	Array Size	Intrinsic
SIZE(A,M) or SIZE(A$,M)	Dimension Size	Intrinsic
SNG(X)	Real Single	Intrinsic
SQR(X)	Square Root	Intrinsic
SRCH(A,X[,M])	Numeric Search	Intrinsic
SRCH(A$,C$[,M])	Character Search	Intrinsic
SREP$(C$,M,D$,E$)	Search and Replace	Intrinsic
STR$(X)	String Conversion	Intrinsic
SUM(A)	Sum	Intrinsic
TAN(X)	Tangent	Intrinsic
TANH(X)	Hyperbolic Tangent	Intrinsic
TIME	Time in Seconds	Intrinsic
TIME$	Time (HH:MM:SS)	Intrinsic
TRN(A) or TRN(A$)	Transpose	Array
TRUNCATE(X,M)	Truncate	Intrinsic
UDIM(A,M) or UDIM(A$,M)	Upper Dimension	Intrinsic

Function Name	Purpose	Category
UPRC$(C$)	Uppercase	Intrinsic
VAL(C$)	Value Conversion	Intrinsic
ZER	Zero	Array

The intrinsic and array functions listed above are described in the following section, in alphabetical order. For each intrinsic function the description includes syntax and usage rules, plus an example. Complete descriptions of the array functions are found in "Array Functions" on page 286. See also "Functions" on page 43.

Descriptions of Intrinsic Functions and Array Functions

Note: Unless otherwise noted, the examples in the following section assume the ARITHMETIC DECIMAL and SPREC options are in effect. Results may differ with ARITHMETIC NATIVE and/or LPREC.

ABS(X) Absolute Value

Returns the absolute value of X. The argument type is numeric (integer, decimal, or real). The type of the result is the same as the type of the argument.

Example

```
ABS(-1.2) returns the value 1.2
ABS(3.4)  returns the value 3.4
```

ACOS(X) Arccosine

Returns the arccosine (the inverse function of the cosine) of X. X can be integer, decimal, or real. X must be in the range -1 to 1. The result is radians in the range 0 to PI. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
ACOS(0) returns the value 1.5707963267948966
```

AIDX(A) or AIDX(A$) Ascending Index Array Function

See "Ascending Index Array Function (AIDX)" on page 287

Related Functions

- "Descending Index Array Function (DIDX)" on page 289
- "Ascending Sort Array Function (ASORT)" on page 288
- "Descending Sort Array Function (DSORT)" on page 291

ANGLE(X,Y) Angle

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, decimal, or real. The result type is determined by the argument with the highest type; the priority of numeric types, from highest to lowest, is decimal, real double, real single, and integer. If this type is real or decimal, the result type will be the same as that type. If, however, both arguments are integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
100 PRINT ANGLE(-1,0)
```

displays the value 3.14159.

Related Functions

- "DEG(X) Radians to Degrees" on page 389

ASIN(X) Arcsine

Returns the arcsine (the inverse function of the sine) of X. X must be in the range -1 to 1 and may be integer, decimal, real single, or real double. The result is radians in the range -PI/2 to PI/2. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
ASIN(1) returns the value 1.5707963267948966
```

ASORT(A) or ASORT(A$) Ascending Sort Array Function

See "Ascending Sort Array Function (ASORT)" on page 288.

Related Functions

- "Descending Sort Array Function (DSORT)" on page 291
- "Ascending Index Array Function (AIDX)" on page 287
- "Descending Index Array Function (DIDX)" on page 289

ATN(X) Arctangent

Returns the arctangent (the inverse function of the tangent) of X. X may be integer, decimal, or real. The result is radians in the range -PI/2 to PI/2. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
ATN(1) returns the value 0.78539816339744829
```

CEIL(X) Ceiling

Returns the smallest integer greater than or equal to X. The argument type is numeric (integer, decimal, or real). The result type is the same as the argument type.

Example

```
CEIL (-1.2) returns the value -1.0
CEIL (2.3)  returns the value  3.0
```

Related Functions

- "INT(X) Integer" on page 396

CEN(X) Fahrenheit to Centigrade

Returns the degrees Centigrade corresponding to X degrees Fahrenheit using the formula (X-32)*5/9. X can be integer, decimal, or real, and must be greater than or equal to -459.67 (absolute zero, Fahrenheit). If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
CEN(32) returns the value 0.0
CEN(212) returns the value 100.
```

Related Functions

- "FAH(X) Centigrade to Fahrenheit" on page 392

CHR$(M) Character

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 the COLLATE option (see "OPTION Statement" on page 312 and Appendix C, "Character Set Collating Sequences" on page 505).

Example: OPTION COLLATE STANDARD (ASCII)

```
CHR$(53) returns the value "5"
CHR$(65) returns the value "A"
```

Example: OPTION COLLATE NATIVE (EBCDIC)

```
CHR$(194) returns the value "B"
CHR$(241) returns the value "1"
```

Related Functions

- "ORD(C$) Ordinal Position" on page 403

CNT Count

Returns the number of data items successfully processed by the last input/output (I/O) statement executed. If the INPUT or INPUT File statement was executed and data input was terminated with a slash (/) to indicate end of data, CNT returns the number of data items processed prior to the slash character. The result type is integer.

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.

Note: If the CNT function is referenced while executing an I/0 statement, the value returned will be the number of data items processed by the statement at the time the function is referenced.

Example

```
100 PRINT CNT;CNT;CNT;CNT
```

will print

```
0 1 2 3
```

CODE Code

For the last exception which caused a transfer of control (via an ON ERROR GOTO statement or an equivalent exception-recovery clause in an I/O statement), CODE returns the integer representation of the status/error code forwarded by the host operating system or another product for exceptions occurring outside of BASIC. Such exceptions can occur when the program performs I/O using host system services (for example, VSAM), requests execution of host system commands (for example, CALL SYSTEM), communicates with subprograms written in other languages (for example, CALL PLI), invokes functions from another product (for example, CALL GDDM), or accesses the relational data system (for example, CALL SQL).

CODE has a result type of integer. The scope of CODE is local to the program unit where the exception occurred. The value of CODE is maintained in synchronization with the ERR and LINE intrinsic functions. Thus if a subsequent exception causes a transfer of control, and the exception does not involve the host system, the value of CODE is reinitialized to zero.

Note: CODE is often used in conjunction with the ERR and LINE intrinsic functions to determine the category of the exception and BASIC statement where

the exception occurred. Since the value of CODE depends on the particular host system service or product function generating the exception, it is highly system dependent. Also, in order to determine the nature of certain system exceptions, the hexadecimal representation of the integer CODE value may be required. For details, see your IBM BASIC System Services manual.

Example

```
 * LIST
100 ON ERROR GOTO error_handler
110 CALL GDDM("fsshow","xxxxxxxx")   ! file "xxxxxxxx" does not exist
120 STOP
130 error_handler:                   ! Error message for exceptions
140 PRINT "Exception "; err; " at line "; line; " System code =";code
150 END
 * RUN
Exception -11015  at line  110  System code = 307
END AT LINE 150.
```

In this example, it is assumed that the Graphical Data Display Manager program product (GDDM) is accessible to BASIC, and that the GDDM display file named "xxxxxxxx" does not exist. Therefore, at statement 110, GDDM is unable to complete the "fsshow" . function, and returns to BASIC in error. This causes the following to occur:

- The value of the ERR intrinsic function is set to -11015 (BASIC exception - "System error returned on CALL GDDM").
- The LINE intrinsic function is set to the line number where the exception occurred (line 110).
- The CODE intrinsic function is set to the error number returned to the host system by GDDM (error 307 - "File not found").

Related Functions

- "ERR Exception Code" on page 391
- "LINE Line Number" on page 399

CON Constant Array Function

See "Constant Array Function (CON)" on page 289.

COS(X) Cosine

Returns the cosine of X, where X is in radians and ABS(X) < PI*(2**50). The argument type may be integer, decimal, or real. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
COS(PI) returns the value -1.0
```

COSH(X) Hyperbolic Cosine

Returns the hyperbolic cosine of X. X can be integer, decimal, or real. The absolute value of X must be less than 175.366. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
COSH(0) returns the value 1.0
```

COT(X) Cotangent

Returns the cotangent of X. X can be integer, decimal, or real. X is in radians, with an absolute value less than PI*(2**50). If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
COT(PI/4) returns the value 1.0
```

CSC(X) Cosecant

Returns the cosecant of X, where X is in radians and ABS(X) <PI*(2**50). The argument type may be integer, decimal, or real. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
CSC(PI/2) returns the value 1.0
```

DAT$[(M)]

"YYYY/MM/DD"

Returns the Gregorian date in the character representation "YYYY/MM/DD." Its argument is a Julian date and is numeric (numeric values are rounded to integer if necessary). Its result type is character. If the argument is omitted, DAT$ returns today's Gregorian date. If the argument is omitted and there is no date available from the system, the value 0000/00/00 is returned.

By definition, Julian day 0 corresponds to 1 January, 4713 B.C. The Julian date is the number of days which have elapsed since then. Below are valid ranges of the argument (Julian date) and corresponding values (Gregorian date) returned:

```
Julian:      1721120    to   5373484     (numeric)
Gregorian: "0000/03/01" to "9999/12/31" (character)
```

Example

```
DAT$(2369916) returns the value "1776/07/04"
```

Related Functions

- "JDY[(C$)] Julian Date" on page 397

DATE YYDDD

Returns the current date in the integer 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 date available from the system, the value returned by DATE is -1. DATE has no argument. The result type is integer.

Example

```
100 LET N = DATE
```

If the current date is January 30, 1987, N is set to 87030.

DATE$

"YY/MM/DD"

Returns the current date in the character representation "YY/MM/DD." If there is no date available from the system, the value of DATE$ will be "00/00/00." DATE$ has no argument. The result type is character.

Example

```
100  LET A$ = DATE$
```

If the current date is January 30, 1987, A$ is set to "87/01/30." If there is no date available from the system, A$ is set to "00/00/00."

DBL(X) Real Double

Converts X to real double. The argument is numeric (integer, decimal, or real). Decimal values are rounded, if necessary. The result type is real double.

Example

```
100 INTEGER I
110 I = 8
120 CALL DSUB (DBL(I))
130 END
140 SUB DSUB (A)
150 REAL DOUBLE A
160 PRINT A
170 END SUB
```

In this example, the subprogram requires a real double argument. In order to pass the value of I, the value of I must be converted.

Related Functions

- "SNG(X) Real Single" on page 412
- "REAL(X) Real" on page 406

DEC(X) Decimal

Converts X to decimal. The argument type is numeric (integer, decimal, or real). The result type is decimal.

Example

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

```
100 INTEGER I
110 I = 3
120 CALL DSUB (DEC(I))
130 END
140 SUB DSUB(B)
150 DECIMAL B
160 PRINT B
170 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) Radians to Degrees

Returns the number of degrees of X, where X is in radians. X can be integer, decimal, or real. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
DEG(1) returns the value 57.295779513082321
DEG(2*PI) returns the value 360
```

Related Functions

- "RAD(X) Degrees to Radians" on page 405

DET[(A)] Determinant

Returns the value of the determinant of the square numeric array A. The argument (when present) is numeric. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46. If the argument is omitted, DET returns the determinant of the last array inverted using the INV function in a MAT statement, and the result type depends the context of the function. (See "Special Cases of Numeric Result Type" on page 45.) If the argument is *not* a square matrix, an exception occurs.

Example

```
 * LIST
100 DIM ARRAY(1,1)
110 ARRAY(0,0)=0: ARRAY(0,1)=1
120 ARRAY(1,0)=2: ARRAY(1,1)=3
130 MAT PRINT ARRAY
140 NEW=DET(ARRAY)
150 PRINT 'THE DETERMINANT IS',NEW
160 END
 * RUN
 0                    1

 2                    3

THE DETERMINANT IS  -2
END AT LINE 160.
```

Related Functions

- "Inverse Array Function (INV)" on page 293

DIDX(A) or DIDX(A$) Descending Index Array Function

See "Descending Index Array Function (DIDX)" on page 289.

Related Functions

- "Ascending Index Array Function (AIDX)" on page 287
- "Ascending Sort Array Function (ASORT)" on page 288
- "Descending Sort Array Function (DSORT)" on page 291

DOT(A,B) Dot Product

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, decimal, or real. The result type is the same as the argument with the higher type; the priority of numeric types, from highest to lowest, is decimal, real double, real single, and integer.

Example

```
 * LIST
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
END AT LINE 700.
```

DSORT(A) or DSORT(A$) Descending Sort Array Function

See "Descending Sort Array Function (DSORT)" on page 291.

Related Functions

- "Ascending Sort Array Function (ASORT)" on page 288
- "Ascending Index Array Function (AIDX)" on page 287
- "Descending Index Array Function (DIDX)" on page 289

EPS Epsilon

The Greek letter Epsilon (ε) is traditionally used in calculus to denote an arbitrarily small number (as close to zero as desired). Such a small number is useful for numeric approximations of continuous phenomena.

EPS returns the smallest positive number the type can represent. The result type depends on the context in which the function is used. (See "Special Cases of Numeric Result Type" on page 45.) EPS has no arguments.

The smallest decimal number is .1E-999. The smallest real single and real double number is approximately .53976053469340279E-78.

Example

```
 * LIST
10 DECIMAL X
20 X = EPS
30 PRINT X
40 END
 * RUN
1.E-1000
END AT LINE 40.
```

Related Functions

- "INF Infinity" on page 396

ERR Exception Code

Returns the exception code of the last exception which caused a transfer of control. Exception codes are listed in Appendix A, "Exception Codes" on page 493. ERR returns a result type of integer and has an initial value of zero. Its scope is local to the program unit where the exception occurred. See "Exception Handling Statements" on page 127.

Example

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

Related Functions

- "CODE Code" on page 385
- "LINE Line Number" on page 399

EXP(X) Exponential Value

Returns the exponential value of X (the value of the base of natural logarithms):

```
e = 2.7182818284590451
```

raised to the power X. If X is less than the smallest number the installation allows, its value is replaced by 0, and the value returned will be 1.0. X can be integer, decimal, or real. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

If the result type is real, X must be less than or equal to (approximately) 174.673. If the result type is decimal, X must be less than or equal to 2300.2825079105. If X exceeds these limits, an overflow occurs.

Example

```
PRINT EXP(1)
```

will display

```
2.71828
```

Related Functions

- "LOG(X) Natural Logarithm" on page 399

FAH(X) Centigrade to Fahrenheit

Returns the degrees Fahrenheit corresponding to X degrees Centigrade (Celsius), using the formula (X * 9/5) + 32. X can be integer, decimal, or real, and must be greater than or equal to -273.15, which is absolute zero in Centigrade. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
FAH(0) returns the value 32.0
FAH(100) returns the value 212.
```

Related Functions

- "CEN(X) Fahrenheit to Centigrade" on page 384

FILE(M) File Status

Returns a numeric value indicating the current status of the file specified by the file reference number M. The result type is integer, and the status value is reset after each I/O operation.

```
Numeric
Value   Status Explanation

 -1     File is not open or M is less than 1 or greater than 255.
  0     File is open (last I/O operation occurred successfully)
  1     Last I/O operation was an OPEN statement which opened file
        with default access of INPUT
  3     Last I/O operation was an OPEN statement which opened file
        with default access of OUTIN
 10     End-of-file exception occurred during last operation-input*
 11     End-of-file exception occurred during last operation-output*
 20     Transmission error occurred during last operation-input*
 21     Transmission error occurred during last operation-output*

 * = For this function, RESET is considered an input operation, and
     DELETE is considered an output operation.
```

Example

```
 * LIST
100 PRINT File(1)
110 OPEN #1: "newfile", ACCESS OUTPUT, TYPE INTERNAL
120 PRINT File(1)
130 WRITE #1: "abcdefg"
140 PRINT File(1)
150 CLOSE #1
160 PRINT File(1)
170 OPEN #2: "newfile", TYPE INTERNAL
180 PRINT File(2)
190 OPEN #3: "ntvfile", TYPE NATIVE
200 PRINT File(3)
210 END
 * RUN
-1
 0
 0
-1
 1
 3
END AT LINE 210.
```

The first file status value is -1 indicating that file #1 is not open. Once the file is opened at line 110, the status is 0 indicating that the OPEN statement was successful, and that an access attribute was explicitly specified (that is, access was not defaulted). The file status is also 0 after the successful write operation at line 130. When the file is closed at line 150, the file status becomes -1, since the file is no longer open.

The OPEN for file #2 at line 170 results in a file status of 1 because the access attribute was defaulted to INPUT. Likewise, the OPEN for file #3 at line 190 produces a status of 3 due to the default access of OUTIN.

Notes on Use

The FILE intrinsic function is most often used to determine whether or not a file is currently open (that is, FILE values of -1 or 0). Status values greater than 0 are supported in order to provide compatibility with other BASIC implementations; however, generally speaking, these additional values are unnecessary and their use is not encouraged. Status values of 1 and 3 are seldom needed since explicit specification of the access attribute in the OPEN statement is a common (and highly recommended) practice. Likewise, the error status values (10, 11, 20, 21) are little used due to BASIC's exception handling facilities (see "Exception Handling Statements" on page 127).

FILENUM File Number

Returns the file reference number (1 to 255) of the file in which an exception has occurred. Additional data about the cause of the exception can be obtained through the use of the function FILE(M). The name of the file can be obtained through the use of the function FILE$(M). The result type is integer.

If no exception 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 1 when the READ File statement transfers control to 400 on end-of-file.

FILE$(M) File Name

Returns the filename associated with the file reference number M. The argument type is numeric (numeric values are rounded to integer if necessary). The result type is character. The null string is returned if the file is not open, or if M is less than 1 or greater than 255.

Example

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

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

FP(X) Fractional Part

Returns the sign of X times the fractional part of the absolute value of X. The argument is of type numeric (integer, decimal, or real). The result will be between -1 and +1 if the argument has a fractional part. The result will be 0 if the argument has no fractional part. The result type is the same as the type of the argument.

Examples

```
FP(-1.2) returns the value -.2
FP(3.4)  returns the value  .4
FP(400)  returns the value  0
```

Related Functions

- "IP(X) Integer Part" on page 397

IDN Identity Array Function

See "Identity Array Function (IDN)" on page 291.

IFIX(X) Rounded Integer Value

Returns the rounded integer value of X. The argument type can be integer, decimal, or real. The result type is integer.

IFIX(X) is equivalent to SGN(X) * INT (ABS(X) + 0.5), except that the result type is always integer.

Examples

```
IFIX (3.5)  returns the value  4
IFIX (3.2)  returns the value  3
IFIX (-3.5) returns the value -4
IFIX (-3.2) returns the value -3
```

Note: BASIC uses an algorithm similar to the IFIX definition when rounding variable assignments, evaluating expressions, converting data types, and formatting with the PRINT statement. This algorithm always "rounds down" when given a negative argument with a decimal fraction exactly equal to .5 (see third example, above). The ROUND function should be used when "rounding up" is desired.

Related Functions

- "INT(X) Integer" on page 396
- "IP(X) Integer Part" on page 397
- "ROUND(X,M) Round" on page 408

INF Infinity

Returns the largest positive number the type can represent. The result type depends on the context in which the function is used. (See "Special Cases of Numeric Result Type" on page 45.)

Example

```
 * LIST
10 DECIMAL X
20 X = INF
30 PRINT X
40 END
 * RUN
1.E+999
END AT LINE 40.
```

The largest decimal number is .99999999999999999E+999 (note that this value is rounded when printed, as shown in the above example).

The largest real double number is approximately .72370055773322621E+76.

The largest real single number is approximately .72370051459731155E+76.

Related Functions

- "EPS Epsilon" on page 391

INT(X) Integer

Returns the largest integer value less than or equal to X. This value is commonly referred to as the floor of X. The argument must be numeric (integer, decimal, or real). The result type is the same as the type of the argument.

Examples

```
INT(1.3)  returns the value  1
INT(-1.3) returns the value -2
```

Related Functions

- "CEIL(X) Ceiling" on page 384
- "IFIX(X) Rounded Integer Value" on page 395
- "IP(X) Integer Part" on page 397

INV(A) Inverse Array Function

See "Inverse Array Function (INV)" on page 293.

Related Functions

- "DET[(A)] Determinant" on page 389

IP(X) Integer Part

Returns the integer part of the absolute value of X times the sign of X. The argument must be numeric (integer, decimal, or real). The result type is the same as the type of the argument.

IP(X) is equivalent to SGN(X) * INT (ABS(X)).

Examples

```
IP(-1.2) returns the value -1
IP(3.4)  returns the value  3
```

Related Functions

- "FP(X) Fractional Part" on page 395
- "IFIX(X) Rounded Integer Value" on page 395
- "INT(X) Integer" on page 396

JDY[(C$)] Julian Date

Returns the Julian date for the corresponding Gregorian date. If the argument is omitted, the current date is returned. If the argument is omitted and there is no date available from the system, the value 0 is returned. The argument is character. The result type is integer.

The argument for JDY (C$) must be of 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.

By definition, Julian day 0 corresponds to January 1, 4713 B.C. The Julian date is the number of days that have elapsed since that date.

Example

```
100 J = JDY ('1960/01/01')
```

J is set to 2436935.

Related Functions

- "DAT$[(M)] " on page 387

KEYNUM Key Number

Returns the number of the PF key which caused the SKEY condition. (See "ON Condition Statement" on page 298.) If an SKEY exception has not occurred, zero is returned. The result type is integer.

Example

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

If, for example, the user pressed PF3 when input was requested at line 110, XYZ would be assigned the value 3 at line 980.

KLN(M) Key Length

Returns the length of the embedded key (stated in bytes) for the file specified by file reference number M. The argument type can be integer, decimal, or real. The result type is integer. If the file is not keyed, or is not currently open, or if M is less than 1 or greater than 255, a value of -1 is returned.

Example

```
100 I = KLN(1)
```

I is assigned the number of bytes in the key associated with file #1.

KPS(M) Key Position

Returns the byte position for the start of the embedded key for the file specified by the file reference number M. The argument type can be integer, decimal, or real. The result type is integer. If the file is not keyed, or is not currently open, or if M is less than 1 or greater than 255, a value of -1 is returned.

Example

```
1000 LET I = KPS(2)
```

I is assigned the integer value of the starting byte position of the key for records in file #2.

LEN(C$) Length

Returns the number of characters in the value associated with C$. The argument type is character. The result type is integer.

Example

```
100 A$ = 'THIS'
110 LN = LEN(A$)
```

LN is set to 4.

```
200 B$ = ''
210 LN = LEN(B$)
```

LN is set to 0.

LINE Line Number

Returns the line number of the most recent statement whose execution caused a transfer of control caused by an exception. LINE returns a result type of integer and has an initial value of zero. Its scope is local to the program unit where the exception occurred. See "Exception Handling Statements" on page 127.

Example

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

Related Functions

- "ERR Exception Code" on page 391
- "CODE Code" on page 385

LOG(X) Natural Logarithm

Computes the natural logarithm of the number represented by X.

X can be integer, decimal, or real. The natural logarithm is only defined for positive numbers; therefore, zero and negative values of X result in an exception. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
LOG(2) returns the value 0.69314718055994533
```

Related Functions

- "EXP(X) Exponential Value" on page 392

LOG2(X) Base 2 Logarithm

Computes the base 2 logarithm of the number represented by X. X can be integer, decimal, or real. The base 2 logarithm is only defined for positive numbers; therefore, zero and negative values of X result in an exception. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
LOG2(2) returns the value 1.0
```

LOG10(X) Common Logarithm

Computes the common logarithm (base 10) of the positive number represented by X.

X can be integer, decimal, or real. The base 10 logarithm is only defined for numbers greater than zero; therefore, zero and negative values of X result in a exception. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
LOG10(2) returns the value 0.3010299956639812
```

LPAD$(C$,M) Left Pad

If M is greater than the current length of C$, LPAD$ returns the string of M characters produced by adding spaces to the left of C$. If M is less than current length of C$, LPAD$ returns the value of C$. The argument C$ is character, and the argument M is numeric and rounded to integer. The result type is character. This function is useful for right justifying character data for use in report headings, for example.

Example

```
500 LET B$ = LPAD$("ABC",5) ! B$ is set to "  ABC"
```

There are two spaces added to the left of "ABC".

Related Functions

- "RPAD$(C$,M) Right Pad" on page 410

LTRM$(C$) Left Trim

Returns the value of string C$ with all leading space characters removed. Both the argument and result types are character.

Example

```
200 B$ = LTRM$("  ABC") ! B$ is set to "ABC"
```

The leading spaces are removed (trimmed off).

Related Functions

- "RTRM$(C$) Right Trim" on page 410

LWRC$(C$) Lowercase

Returns the string of characters resulting from replacing each uppercase letter in the string C$ by its lowercase counterpart. Both the argument and result types are character.

Example

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

B$ is set to "abc*."

Related Functions

- "UPRC$(C$) Uppercase" on page 417

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

Returns the largest (maximum) of its numeric (integer, decimal, or real) arguments. If any one of the numeric arguments is decimal, the result is decimal. If any one of the numeric arguments is real double (and none are decimal), the result is real double. If any one of the numeric arguments is real single (and none are decimal or real double), the result is real single. If all the arguments are integer, the result is integer.

Example

If A=2 , B=5, and C = 7, the statement:

```
200 LET D = MAX(A,B,C)
```

assigns D a value of 7,

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

assigns C a value of 1.2234.

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

Returns the smallest (minimum) of its numeric (integer, decimal, or real) arguments. If any one of the numeric arguments is decimal, the result is decimal. If any one of the numeric arguments is real double (and none are decimal), the result is real double. If any one of the numeric arguments is real single (and none are decimal or real double), the result is real single. If all the arguments are integer, the result is integer.

Example

If A=2, B=5, and C=7; the statement:

```
300 LET D = MIN(A,B,C)
```

assigns D a value of 2.

MOD(X,Y) Modulo

Returns X modulo Y, which is a number in the range of 0 to Y-1, representing the relationship of Y to X, where Y is a modulus. If Y is nonzero, MOD(X,Y)=X-Y*INT (X/Y). If Y is zero, MOD(X,Y)=X. The argument type is numeric (integer, decimal, or real). The result type is the same as the type of the argument with the highest type; the priority of numeric types, from highest to lowest, is decimal, real double, real single, and integer.

If Y is zero, a zero divide (ZDIV) exception occurs. The SYSTEM action is to display a message and continue execution using X as the value of the function.

Examples

```
MOD(17,3)    returns the value  2
MOD(-17,3)   returns the value  1
MOD(17,-3)   returns the value -1
MOD(-17,-3)  returns the value -2
MOD(16,4)    returns the value  0
MOD(5.2, 2)  returns the value  1.2
MOD(3,.7)    returns the value   .2
MOD(6,0)     returns the value  6  (system action after exception)
```

Related Functions

- "REM(X,Y) Remainder" on page 407

NUL$ Null String Array Function

See "Null String Array Function (NUL$)" on page 294.

ORD(C$) Ordinal Position

Returns the ordinal position of the character named by the string value of C$ 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 C$ are the single character graphics of the character set and the 2- and 3-character mnemonics of that set. The acceptable values for both character sets are shown in Appendix C, "Character Set Collating Sequences" on page 505. The argument type is character. The result type is integer. The result depends on the current setting of OPTION COLLATE (see "OPTION Statement" on page 312).

Example

For the ASCII character set (OPTION COLLATE STANDARD).

```
ORD("BS")=8
ORD("A")=65
ORD("5")=53
ORD("SOH")=1
```

Related Functions

- "CHR$(M) Character" on page 384

PARM$ Parameter

Returns a character string that has been passed to the program. When executing inside the BASIC environment, you can pass a character string to your program by using the PARM clause of the RUN command. For details on how to pass a character string when executing outside the BASIC environment, see your IBM BASIC System Services manual.

When a parameter is specified with the RUN command, leading and trailing spaces are removed from the value (see "RUN Command" on page 472).

PARM$ returns a null string if no parameter was specified or if the parameter consists entirely of spaces.

Example

If you execute the program called EMPLYEE with the command:

```
* RUN EMPLYEE (PARM 'OLSEN')
```

the statement:

```
100 LET NAME$ = PARM$
```

will assign the character string "OLSEN" to NAME$.

PI

Returns the constant pi (π), which is mathematically defined as the quotient of the circumference of a circle divided by its diameter. In BASIC, PI has the following values:

```
The decimal value for PI is                      3.1415926535897932
The real double value for PI is approximately    3.1415926535897931
The real single value for PI is approximately    3.1415929794311523
```

The result type depends on the context in which the function is used. (See "Special Cases of Numeric Result Type" on page 45.)

Example

```
 * LIST
100 R=10
110 AR=PI*R**2
120 PRINT AR
130 END
 * RUN
314.159
END AT LINE 130.
```

POS(C$,D$) Position

Returns the position of the first occurrence of the character value of D$ within the value of C$. If D$ does not occur within C$, POS(C$,D$) returns zero. POS(C$,"") is defined as one. ("" is the null character.) The argument types are character. The result type is integer.

Example

```
490 A$ = " 123-4.56" : B$ = "-"
500 LET X = POS(A$,B$)
```

X is assigned the value 5.

```
600 LET X = POS("LEARN YOUR ABCS", "ABC")
```

X is set equal to 12

```
700 X = POS("ABC","123")
```

X is assigned the value zero, because "123" does not occur in "ABC".

POS(C$,D$,M) Position

Returns the position of the first occurrence of the character value of D$ within the value of C$ on or after (to the right of) character position M in C$. If D$ does not exist within the designated portion of C$, or if M is greater than the length of C$, the value returned is zero. If M is less than 1, 1 is used in place of M. POS(C$,"",M) is defined as M, as long as M is not greater than the length of C$.

The arguments C$ and D$ are character; M may be any numeric type and is rounded to integer. The result type is integer.

Example

If A$ has the value "UNDERSTANDING", then in the statement:

```
300 LET X = POS(A$,"ND",1)
```

X is assigned the value 2.

In the subsequent statement:

```
400 LET Y = POS(A$,"ND",4)
```

Y is assigned the value 9, because the search started after "UND", at character "E".

Finally:

```
500 LET Z = POS (A$,"AN",10)
```

assigns Z the value 0 because "DING" does not contain "ND".

PRD(A) Array Product

Returns the product of the elements of the array specified by A. The argument type must be numeric (integer, decimal, or real). The result type is the same as the type of the argument.

Example

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

ARAPROD is assigned the value 600, which is the product of 4 * 3 * 10 * 5.

RAD(X) Degrees to Radians

Returns the radian equivalent to X degrees. The argument type is numeric (integer, decimal, or real). If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Examples

```
10 DECIMAL
20 X=RAD(23)
30 PRINT USING 40: X
40 : #.################
```

When executed, this program prints X as 0.4014257279586958.

Similarly, RAD(180) in this program equals 3.1415926535897932.

Related Functions

- "DEG(X) Radians to Degrees" on page 389

REAL(X) Real

Converts X to real single if SPREC is in effect (equivalent to using the SNG function) or to real double if LPREC is in effect (equivalent to using the DBL function). The argument is numeric (integer, decimal, or real).

Example

```
100 INTEGER I
110 I = 8
120 CALL RSUB (REAL(I))
130 END
140 SUB RSUB (A)
150 REAL A
160 PRINT A
170 END SUB
```

In this example, the subprogram requires a real argument. In order to pass the value of I, the value of I must be converted to real.

Related Functions

- "DBL(X) Real Double" on page 388
- "SNG(X) Real Single" on page 412

REC(M) Record Number

Returns the number of the last record processed in the relative record file having file reference number M (1 to 255). Returns a zero if no records have been processed or if the file has been reset to the beginning or if the file has been scratched. Returns -1 if the file is closed or is not a relative file, or if M is less than 1 or greater than 255. The argument type is numeric. The result type is integer.

After RESET BEGIN, or SCRATCH, REC(M) returns 0.

After RESET #M, REC=10 REC(M) returns 10.

Example

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

X is set to the number of the last record either read or written in file #1.

REM(X,Y) Remainder

If Y is nonzero, returns the remainder of X divided by Y defined as follows:

```
REM(X,Y)=X-Y*IP(X/Y)
```

This is the difference between the dividend (X) and the product of the divisor (Y) times the integer part of the quotient (X/Y).

If Y is zero, a zero divide (ZDIV) exception occurs. The SYSTEM action is to display a message and continue execution using X as the value of the function.

The argument types are numeric (integer, decimal, or real). The result type is the same as the type of the argument with the highest type; the priority of numeric types, from highest to lowest, is decimal, real double, real single, and integer.

Example

```
REM(17,3)     returns the value  2
REM(-17,3)    returns the value -2
REM(17,-3)    returns the value  2
REM(-17,-3)   returns the value -2
REM(16,4)     returns the value  0
REM(5.2,2)    returns the value  1.2
REM(3,.7)     returns the value  .2
REM(6,0)      returns the value  6 (system action after exception)
```

Related Functions

- "MOD(X,Y) Modulo" on page 402

RLN(M) Record Length

Returns the length of the last record referenced for file reference number M. Zero is returned if no records have been referenced. Negative one is returned if the file is closed or if M is less than 1 or greater than 255. The argument type is numeric (rounded to integer if necessary). The result type is integer.

Example

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

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

RND[(X)] Random

Provides the next pseudorandom number in a sequence of pseudorandom numbers uniformly distributed in the range 0 <= RND < 1.

If the argument X is included, RND also assigns the value of X as the seed value for the pseudorandom number generator.

For a particular value of X, RND will always return the same value. The value of X should only be used to start a sequence, not to get the next number in a sequence. The value of X should be greater than 0 and less than 1. If the value of

X is not between 0 and 1, the next number in the current sequence is returned, as if no argument had been specified.

The argument (when present) is numeric. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

If the function is invoked without an argument, the result type depends upon the context of the function. (See "Special Cases of Numeric Result Type" on page 45.)

When a program is executed, the RND function is initialized with a default seed value; every time the program is run the same sequence is used. Use the X argument to change the sequence or use the RANDOMIZE statement (see "RANDOMIZE Statement" on page 343) to start an unpredictable sequence.

The random number sequence is global to the main program and all subprograms and functions.

Example

This program segment shows how pseudorandom numbers can be obtained in various ranges. The last line supplies RND with a seed to change the sequence used by RND.

```
110 PRINT RND                       !  0 LE output LT  1
120 PRINT RND + 1                   !  1 LE output LT  2
130 PRINT 10 * RND + 1              !  1 LE output LT 11
140 PRINT INT(10 * RND +1)          !  Integer  1, 2, 3..10
150 PRINT INT(10 * RND(0.7) + 1)    !  Same as 140, but RND
160                                 !    gets seed 0.7
```

When executed, this program segment produces the output:

```
 .959621
1.34709
6.13322
4
9
```

ROUND(X,M) Round

Returns the value of X, rounded to M decimal places. If M is negative, X is rounded to M places to the left of the decimal point. ROUND(X,M) is equivalent to:

```
INT(X*10**M+.5)/10**M
```

The argument types are numeric (integer, decimal, or real).

The argument M is first rounded to the nearest integer. If the argument X is real or decimal, the result type will be the same as the argument type. If, however, the argument X is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

These three statements show how rounding of the same integer is affected by different numbers of decimal places specified:

```
100 PRINT ROUND(1234, 2)    ! round to nearest .01
110 PRINT ROUND(1234, 0)    ! round to nearest 1
120 PRINT ROUND(1234,-2)    ! round to nearest 100
```

When executed, these statements produce the output:

```
1234
1234
1200
```

The next three statements show how rounding of the same decimal number is affected by different numbers of decimal places specified:

```
140 PRINT ROUND(1234.567, 2)   ! round to nearest .01
150 PRINT ROUND(1234.567, 0)   ! round to nearest 1
160 PRINT ROUND(1234.567,-2)   ! round to nearest 100
```

When executed, these statements produce the output:

```
1234.57
1235
1200
```

The next four statements show the rounding of negative numbers and the special case of decimal fractions exactly equal to .5

```
180 PRINT ROUND ( 3.49, 0)      ! Round to nearest 1
190 PRINT ROUND ( 3.50, 0)      ! Round to nearest 1
200 PRINT ROUND (-3.50, 0)      ! Round to nearest 1
210 PRINT ROUND (-3.51, 0)      ! Round to nearest 1
```

When executed, these statements produce the output:

```
 3
 4
-3
-4
```

Note: The ROUND function "rounds up" and assigns the next greater value whenever the first argument is a decimal fraction ending with the digit 5 (for example, .5, 1.0005, 56.15, -1.655). This occurs regardless of whether the argument is positive or negative. This definition of rounding is different from that used by BASIC when rounding for variable assignment, expression evaluation, data conversion, or formatting with the PRINT statement. In these cases, an algorithm similar to that used by the IFIX intrinsic function is used, and negative decimal fractions ending with 5 are "rounded down" (that is, -.5 is rounded to -1).

Related Functions

- "IFIX(X) Rounded Integer Value" on page 395
- "TRUNCATE(X,M)" on page 416

RPAD$(C$,M) Right Pad

If M is greater than the current length of C$, RPAD$ returns the string of M characters produced by adding spaces to the right of C$. If M is less than the current length of C$, RPAD$ returns the value of C$. The argument C$ is character, and the argument M is numeric and rounded to integer. The result type is character.

Example

```
500 LET B$ = RPAD$("ABC",5) ! B$ is set to "ABC  "
```

There are two spaces added to the right of "ABC".

Related Functions

- "LPAD$(C$,M) Left Pad" on page 400

RPT$(C$,M) Repeat

Repeats the string C$, M number of times. The argument C$ is character, and the argument M is numeric and is rounded to integer. The result type is character.

Example

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

A$ is set to "***".

```
20 B$ = RPT$("/*",4)
```

B$ is set to "/*/*/*/*".

RTRM$(C$) Right Trim

Returns the value of string C$ with all trailing spaces removed (trimmed). Both the argument and result types are character.

Example

```
100 A$="AB CD  "
200 B$=RTRM$(A$) ! B$ is set to "AB CD"
```

Related Functions

- "LTRM$(C$) Left Trim" on page 401

SEC(X) Secant

Returns the secant of X, where X is in radians. X can be integer, decimal, or real, and ABS(X) must be less than PI*(2**50). If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
SEC(PI) returns the value -1
```

SGN(X) Sign

Returns -1 if X is less than zero, zero if X is equal to zero, and +1 if X is greater than zero. The argument type is numeric (integer, decimal, or real), and the result type is integer.

Examples

```
SGN(-2) returns the value -1
SGN(10) returns the value  1
SGN(0)  returns the value  0
```

SIN(X) Sine

Returns the sine of X, where X is in radians. X can be integer, decimal, or real. The absolute value of X must be less than PI*(2**50). If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Examples

```
SIN(3)    returns the value 0.14112000805986738
SIN(PI/2) returns the value 1.0
```

SINH(X) Hyperbolic Sine

Returns the hyperbolic sine of the number X. X can be integer, decimal, or real. The absolute value of X must be less than 175.366. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
SINH(0) returns the value 0.0
```

SIZE(A) or SIZE(A$) Array Size

Returns the number of elements in the array A or A$. The argument A is numeric and the argument A$ is character. The result type is integer.

Example

```
100 DIM A(4),B(4,3)
110 LET N = SIZE(A)
120 LET X = SIZE(B)
```

N is set to 5 and X is set to 20 if the BASE 0 option is in effect.

N is set to 4 and X is set to 12 if the BASE 1 option is in effect.

SIZE(A,M) or SIZE(A$,M) Dimension Size

Returns the number of elements in the Mth dimension of array A or A$. The argument A is numeric, the argument A$ is character, and the argument M is numeric and rounded to integer. The result type is integer.

Example

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

If the BASE 0 option is in effect, N is assigned 11, X is assigned 5, and Y is assigned 4.

If the BASE 1 option is in effect, N is assigned 10, X is assigned 4, and Y is assigned 3.

Related Functions

- "UDIM(A,M) or UDIM(A$,M) Upper Dimension" on page 416

SNG(X) Real Single

Converts X to real single. Numeric values are rounded to real single.

Example

```
100 INTEGER I
110 I = 8
120 CALL RSUB (SNG(I))
130 END
140 SUB RSUB (A)
150 REAL SINGLE A
160 PRINT A
170 END SUB
```

In this example, the subprogram requires a real single argument. In order to pass the value of I, the value of I must be converted to real single.

Related Functions

- "DBL(X) Real Double" on page 388
- "REAL(X) Real" on page 406

SQR(X) Square Root

Returns the square root of X. X must be a positive number; it can be integer, decimal, or real. If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
100 LET X = SQR(9)
110 LET Y = SQR(58)
```

X is set to 3.0 and Y is set to 7.6157731058639082

SRCH(A,X[,M]) or SRCH(A$,C$[,M]) Search

Searches the one-dimensional array A or A$ for the value X or C$ (respectively), optionally beginning the search with element of A or A$ corresponding to subscript M. The value returned is the subscript of the element of A or A$ which first matches X or C$, respectively. If a match is not found, a value of -1 is returned.

The array A or A$ must be one-dimensional. If A is a different type than X, an element of A or X is converted to the higher type before doing the comparison; the priority of numeric types, from highest to lowest, is decimal, real double, real single, and integer. If necessary, M is rounded to an integer value. The result type is integer.

Example

```
100 DIM A(5)
110 FOR X=0 to 5
120   A(X)=X
130 NEXT X
140 VALUE1 = SRCH(A,3)
150 VALUE2 = SRCH(A,2,3)
```

VALUE1 is set to 3 and VALUE2 is set to -1

SREP$(C$,M,D$,E$) Search and Replace

Returns a string whose value is C$, where, starting at position M in string C$, the occurrences of string D$ are located, and those occurrences are replaced with string E$. After replacement occurs, the search continues for another string D$. However, previous characters are not rescanned for D$. The arguments C$, D$, and E$ are character, and the argument M is numeric and rounded to integer. The result type is character. If D$ is not found in C$, the result is the value of C$.

Example

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

D$ is set to "ABC123FG"

STR$(X) String Conversion

Returns a character representation of the numeric value X. The character string is that which would be produced by a PRINT statement displaying the value X. No leading or trailing spaces are included in this representation.

This is the inverse of the VAL function. The argument type is numeric (integer, decimal, or real). The result type is character.

Note that the SPREC or LPREC option in effect will affect the result (see "OPTION Statement" on page 312).

Example

```
STR$(123456789e10) returns "1.23456789E+18" with LPREC
                   returns "1.23457+18" with SPREC
```

Related Functions

- "VAL(C$) Value Conversion" on page 417

SUM(A) Sum

Returns the sum of the elements of a numeric array. The argument is numeric (integer, decimal, or real), and the result type is the same as the argument type.

Example

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

```
100 SUMM=SUM(ARX)
```

SUMM is set to 51.

TAN(X) Tangent

Returns the tangent of X, where X is stated in radians (integer, decimal, or real). The absolute value of X must be less than PI*(2**50). If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
TAN(PI/4) returns the value 1
```

TANH(X) Hyperbolic Tangent

Returns the hyperbolic tangent of the number X (integer, decimal, or real). If the argument is real or decimal, the result type will be the same as the argument type. If, however, the argument is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

Example

```
TANH(0) returns the value 0
```

TIME Elapsed Seconds

Returns the time elapsed since midnight, expressed in seconds. If the time is not available from the system, the value of X is set to -1. There is no argument, and the result type is integer.

Example

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

```
X = TIME
```

X is set to 40500

TIME$ HH:MM:SS

Returns the time of day in 24-hour notation, as the character representation "HH:MM:SS" where HH is hours, MM is minutes, and SS is seconds.

There is no argument, and the result type is character.

If the time of day is not available from the system, the value of A$ is set to "99:99:99."

Example

If the current time is 11:15 P.M.

```
200 LET A$ = TIME$
```

A$ is set to "23:15:00"

TRN(A) or TRN(A$) Transpose Array Function

See "Transpose Array Function (TRN)" on page 294.

TRUNCATE(X,M)

Returns the value of X truncated to M decimal places following the decimal point. The argument types are numeric (integer, decimal, or real). The argument M is rounded to the nearest integer. If X is real or decimal, the result type will be the same as the type of X. If, however, X is integer, the result type depends on the options in effect. See "When the Result Type Depends on the Options in Effect" on page 46.

TRUNCATE(X,M) is equivalent to IP(X * 10**IFIX(M))/(10**IFIX(M))

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

Example

```
100 X=TRUNCATE (PI,4)
110 Y=TRUNCATE (1234.56,-2)
```

X is set to 3.1415
Y is set to 1200

Related Functions

- "ROUND(X,M) Round" on page 408

UDIM(A,M) or UDIM(A$,M) Upper Dimension

Returns the upper limit of dimension M of array A or A$. The argument A is numeric, A$ is character, and M is numeric and rounded to integer. The result type is integer.

Note: The upper limit of a dimension is not necessarily the same value as its SIZE. (See "SIZE(A) or SIZE(A$) Array Size" on page 412 and "SIZE(A,M) or SIZE(A$,M) Dimension Size" on page 412). The SIZE of an array dimension is the number of elements contained in the dimension and depends on the base (starting) dimension index value. This is determined by the OPTION BASE statement. (See "OPTION Statement" on page 312.) Because the highest index value of a given array dimension is independent of the index starting value, the BASE option has no effect on UDIM.

Example

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

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

would cause U to be set to 23.

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

would cause U to be set to 6.

Related Functions

- "SIZE(A) or SIZE(A$) Array Size" on page 412

UPRC$(C$) Uppercase

Returns the string of characters resulting from the replacement of each lowercase letter in the string C$ by its uppercase counterpart. Both the argument and result types are character.

Example

```
UPRC$("abC2") returns "ABC2"
```

Related Functions

- "LWRC$(C$) Lowercase" on page 401

VAL(C$) Value Conversion

Returns the numeric value of C$ if the character string contained in C$ 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 21).

If C$ 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 results in a value which causes an overflow, the usual action for numeric overflow occurs. See "ON Condition Statement" on page 298. If the evaluation of the numeric constant results in a value which causes underflow, then the usual action for numeric underflow occurs. The argument is character type, and the result type depends on the context in which the function is used. (See "Special Cases of Numeric Result Type" on page 45.)

Examples

```
VAL ("123.5")   returns the value 123.5
VAL("  -1.23E45  ") returns  -1.23E+45
VAL ("MCMXVII") causes a CONV exception.
VAL ("123.5XY") causes a CONV exception.
VAL ("2.E-9999") returns the value 0 (system action after
underflow exception)
```

Related Functions

- "STR$(X) String Conversion" on page 414

ZER Zero Array Function

See "Zero Array Function (ZER)" on page 295.

BASIC Commands

This section contains individual discussions of each command. The commands are presented in alphabetic order after the following information. For each command you'll find:

- A brief description
- Format (see "Syntax Notation" on page 143)
- Explanation of items in the format
- A full description
- Error conditions, if any

See also "BASIC Commands" on page 137.

List of Commands and their Minimum Abbreviations

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
PROFILE	PR
PURGE	PU
QUERY	QUE
QUIT	QUI
RENAME	RENA
RENUMBER	RENU
RUN	RU
SAVE	SA
SET LOG	SE LOG
SET MSG	SE MSG

SET OPTION	SE OPTION
SET PF	SE PF
SET PROFILE	SE PROFILE
SET TERM	SE TERM
STORE	ST
SYSTEM	SY

Note: The minimum abbreviation is two characters.

Descriptions of BASIC Commands

AUTO Command

The AUTO command provides automatic line number prompting.

Format

```
AUTO [start-line] [STEP increment]
```

Minimum: AU

where:

start-line
indicates the starting line number. The IBM-distributed default is 100.

increment
is an unsigned, nonzero integer less than 9999999. The IBM-distributed default is 10.

Description

In program line entry mode, BASIC prompts with line numbers so that all you need to do is enter BASIC statements.

The first prompted line number is determined by the following rules:

- If *start-line* is specified, the first prompted line is for *start-line*.
- If *start-line* is not specified and the workspace is empty, the first prompted line is for 100.
- If *start-line* 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.

Subsequent line numbers are derived by adding the STEP increment.

To terminate automatic line number prompting, enter a null line (press the ENTER key after a new line number is displayed). The null line does not become part of the program.

You may also type over the line number and enter a command, immediate statement, or a program line that has a different line number. Any of these actions causes automatic line number prompting to end.

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

```
AUTO 300 STEP 5
```

starts prompting with 300 and increments by 5.

Example

```
AUTO
```

starts prompting at 100 and increments by 10 (if the workspace is empty).

Example

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

```
Format 1
    BREAK [ON] line-num [,line-num]...
Format 2
    BREAK OFF [line-num [,line-num]...]
Format 3
    BREAK?
Format 4
    BREAK ?
Minimum:  BR
```

where:

line-num
is a line number in the workspace.

Description

The BREAK ON command sets breakpoints at the indicated line numbers.

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, and so forth, before continuing execution.

Execution can be resumed, if nothing is done while at the breakpoint, by pressing the ENTER key or by entering the GO command.

If commands that do not alter the workspace program, or if immediate statements have been executed, the GO command may be used to resume execution.

If a command that alters the workspace program or if 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.

A breakpoint is automatically deleted if:

- The line associated with the breakpoint is deleted.
- A MERGE command replaces the line associated with the breakpoint.

All breakpoints are deleted as part of FETCH, INITIALIZE and LOAD commands. (The FETCH will reestablish any breakpoints in effect at the time of the STORE.)

The current line setting remains unchanged for a BREAK command.

Example

If a breakpoint is set on line 100 as follows:

```
BREAK ON 100
```

Then:

- 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.

Example

```
BREAK ON 200,300
```

sets breakpoints at lines 200 and 300.

Example

```
BREAK OFF 200
```

removes a breakpoint previously set for line 200. A break previously set for line 300 (previous example) remains in effect.

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 2 of the CHANGE command displays a line in the terminal input area. It is discussed following Format 1.)

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**
: are line numbers which specify the scope of the command. They may also be specified as 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* starts with a letter (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* in a line are to be replaced by *new-string*.

Description

The CHANGE command specifies lines in the workspace to be changed. All occurrences of *old-string* in each line can be replaced by *new-string* (with 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 and the following space 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 segments of a line.

Although the BASIC language does not distinguish between upper and lowercase (except in character strings), the CHANGE command changes only strings which exactly match *old-string*.

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 space character following the line number (an error occurs if ALL is specified with a null *old-string*).

If *old-string* cannot be located, "STRING NOT FOUND" is displayed.

When any change occurs, the new line(s) is displayed. If more than one line is changed, 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 line segment of a line being 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 line segments of a continued line may be changed. This may result in syntax errors. Notice that all the continuation segments between one line number and the next are associated with the preceding line number, irrespective of the line segments' contents. This means you can inadvertently remove a leading or trailing continuation ampersand and still have the following continuation segments.

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

```
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

```
CHANGE 100 /DAYS/WEEKS/ALL
```

Every occurrence of DAYS, on line 100, is changed to WEEKS.

Example

```
CHANGE 100 TO 200/DAYS/WEEKS/ALL
```

All occurrences of DAYS, between lines 100 and 200 inclusive, are changed to WEEKS.

Example

```
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

```
CHANGE //DAYS/
```

Inserts the word DAYS following the space after the line number of the current line.

Example

```
CHANGE FIRST TO LAST/DAYS//ALL
```

Deletes all occurrences of DAYS from the workspace.

Example

```
CHANGE F TO 400/DAYS/WEEKS/
```

Changes the first occurrence of DAYS to WEEKS in each line from the first line in the workspace through the line numbered 400.

Example

```
CHANGE 400 TO LAST/DAYS/WEEKS/A
```

Changes every occurrence of DAYS to WEEKS found from line 400 through the last line in the workspace.

Example

```
CH OLD NEW
```

Changes the first occurrence of OLD to NEW in the current line.

CHANGE Command—Format 2

There are two formats of the CHANGE command. Format 1 changes character strings within the statements in your workspace. This section discusses the second format, which may be used only with a 327x type terminal while in SCROLL mode. It displays a line in the terminal input area for you to modify.

Format

```
CHANGE [line-number[.segment-number]]
```

Minimum: CH

where:

line-number
identifies an existing line in the workspace. It is an unsigned integer.

segment-number
identifies a particular segment of a multisegment line (a continuation line). It is an unsigned integer.

Description

The second form of the CHANGE command does not perform string replacement. It displays a particular line segment in the 327x terminal input area that can then be modified and reentered.

If you specify a *line-number*, the line number segment of that line is displayed in the input area of the screen. (If a line is not continued, the complete line is the line number segment.)

If you do not specify a *line-number*, the line number segment of the current line is displayed in the input area of the screen.

The segment number is needed only if you are dealing with continued lines and want to change a particular segment of a line. Individual segments of a line are numbered starting with zero. Thus, line 250 itself can be considered line 250.0, and the second continuation segment (the third segment of line 250) is 250.2.

If the line number of a line number segment is changed or deleted, or the leading ampersand of a continuation segment is deleted, the input is treated as if it was entered normally (not in response to a CHANGE command).

As discussed for Format 1 of CHANGE, a trailing ampersand may be altered and the change will take place (this may cause a syntax error).

An error message is displayed if the terminal (for example, a hard copy terminal) does not support Format 2 of the CHANGE command, or if SET TERM LINE is in effect (see "SET TERM Command" on page 488).

The current line is reset to the line accessed by the CHANGE command.

Example

If your program contains the line

```
200 LET DAYS = 52
```

you can recall this line to the input area of 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

If your program contains the line

```
100 OPEN #5: NAME "MYFILE" &
  &  ,ACCESS INPUT         &
  &  ,TYPE NATIVE          &
  &  ,ORGANIZATION KEYED   &
  &  ,RECORDS FIXED        &
  &   EXIT 9000
```

The third segment (the one containing TYPE) can be brought to the screen for editing by

```
CHANGE 100.2
```

Suppose you really wanted the second segment (the one containing ACCESS), you could type over the displayed segment from the previous CHANGE,

```
CHANGE 100.1
```

and bring up the desired segment.

COMPILE Command

The COMPILE command translates a BASIC source program into an object program. You may use the COMPILE command to compile a program currently in the workspace or a program residing in an external file.

Format

```
COMPILE [prog-name] [OBJ[ECT] (obj-name)] [OUT (list-name)] [[OPTIONS]([option-list])]
Minimum:  COM
```

where:

prog-name, obj-name, and list-name
are file specifications which identify the program to be compiled, the object text file, and the listing file, respectively. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

option-list
is a list of compiler options to be used. The choices are given below.

The OBJECT, OUT, and OPTIONS clauses may appear in any order. If the COMPILE command terminates with a right parenthesis, the right parenthesis may be omitted.

Description

Prog-name identifies a file consisting of the BASIC program to be compiled. *Obj-name* and *list-name* identify the files for the OBJECT and OUT clauses, respectively.

When you do not specify an external file (*prog-name*) to be compiled, the program in the workspace will be compiled.

When you specify an external file (*prog-name*) to be compiled, it is loaded into the workspace, the name of the workspace is derived from *prog-name*, and the program is compiled. Workspace naming rules are system dependent (see your IBM BASIC System Services manual). Any program already in the workspace is replaced by the program specified in the COMPILE command.

Specifying an external file functions as if a LOAD command had been issued for that file (except that the syntax messages for the load are not displayed) followed by a COMPILE command (syntax messages are displayed for the compile).

All program units in the workspace are compiled and placed in the same object file as if they were independently compiled and the resulting object files were concatenated into one file.

For dynamic loading of compiled program units, it is recommended that each subprogram be compiled into a separate object file that has the same name as the subprogram. See your IBM BASIC System Services manual for details.

Execution of the COMPILE command and execution of the BASIC compiler outside the BASIC environment is similar. The difference is that the COMPILE command can be used to compile source programs in your workspace.

OBJECT Clause: The OBJECT clause specifies the file on which the object text is to be placed. If omitted (and the OBJECT option is in effect), the object text is placed in your library using a default file specification derived from the workspace name, as described in your IBM BASIC System Services manual.

OUT Clause: The OUT clause directs the listing to a file. If omitted, the listing is placed in your library using a default file specification derived from the workspace name, as described in your IBM BASIC System Services manual.

If either the OBJECT clause (and the OBJECT option is in effect) or the OUT clause is 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. See your IBM BASIC System Services manual for valid file-specifications for the OBJECT and OUT clauses.

COMPILER Options: The following compiler options are available. Each option is a keyword. The keywords must be separated by spaces or commas.

Each option has a default. The defaults, as distributed, are underlined below. The defaults for FLAG, FIPS | NOFIPS, SPREC | LPREC may be changed by using the "SET OPTION Command" on page 483. Compile any program without specifying any compiler options to discover the defaults for your installation.

SOURCE | NOSOURCE

Specifies whether or not the source program lines are to be included in the listing file. This listing includes diagnostic error messages. Most errors are indicated by listing the statement in its original form with the erroneous phrases or characters marked with numbers under the statement. Messages indicating the error type follow and are numbered to correspond to the numbers under the statement. In addition, errors involving the flow of control (for example, branches to undefined statement numbers) are listed at the end of the program. NOSOURCE suppresses printing of lines that do not have syntax messages.

OBJ[ECT] | NOOBJ[ECT]

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 line 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 line 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 an object instruction listing consisting of assembler mnemonics and symbols. 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.

This option may be overridden in a program unit by specifying a value in an OPTION statement (see "OPTION Statement" on page 312).

FIPS | NOFIPS

Specifies whether or not to produce an informational message for any statement found not to conform to the FIPS BASIC syntax. Federal Information Processing Standard (FIPS) BASIC is defined in the publication *Minimal BASIC*, FIPS PUB 68. The FIPS standard is technically equivalent to the *American National Standard for Minimal BASIC*, ANSI X3.60-1978.

When FIPS is specified, these messages are printed even if FLAG(I) is not set.

This option may be overridden in a program unit by specifying a value in an OPTION statement (see "OPTION Statement" on page 312).

SPREC | LPREC

Specifies the maximum number of significant digits to be written by the PRINT statement (without the USING clause) when printing numeric values. SPREC specifies a "short" precision of 6 digits. LPREC specifies a "long" precision of 12 digits.

In addition, SPREC specifies that real data which is defined by REAL statements or by the option ARITHMETIC NATIVE will be real single, and LPREC specifies that such real data will be real double.

This option may be overridden in a program unit by specifying a value in an OPTION statement (see "OPTION Statement" on page 312).

Example

```
COMPILE
```

compiles the source program in your workspace. Defaults will be used for all the options.

Example

```
COMPILE OBJECT (MYOBJECT)
```

compiles the source program in your workspace and writes the object text to a file named MYOBJECT.

Example

```
COMPILE OBJ (MYOBJECT) OUT (LISTFILE) (SOURCE XREF
```

compiles the program in your workspace. It directs the object text to a file named MYOBJECT, and the generated listing to LISTFILE, which will include the source program and cross-reference listings.

Example

```
COMPILE PROGA (XREF, NOOBJ)
```

loads the program named PROGA in the workspace and compiles it. The listing file will include a cross-reference listing. Object text will not be produced.

COPY Command

The COPY command duplicates one or more lines in the workspace.

Format

```
COPY start-line [TO end-line] AT target-line [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*.

target-line
is the target line number.

increment
is an unsigned, nonzero integer less than 9999999. The IBM-distributed default is 1.

The AT and STEP clauses may be interchanged.

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 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.

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 *target-line* to the first copied line; subsequent copied lines are assigned line numbers by adding the STEP *increment* to the previously assigned line number.

STEP Clause: Subsequent copied lines are numbered based on the increment.

This five line example shows how your program might look:

```
100  LET A = 1
110  LET B = 2
120  IF B=2 THEN 100
130  LET D = 4
140  LET E = 5
```

You decide you would like to copy all the lines from the first up to and including line 120. You would like the lines copied to have line numbers starting with 500 and increasing in increments of 50. So, you issue the command:

```
* copy first to 120 at 500 step 50
```

and this is how your program looks after the copy.

```
100  LET A = 1
110  LET B = 2
120  IF B = 2 THEN 100
130  LET D = 4
140  LET E = 5
500  LET A = 1
550  LET B = 2
600  IF B=2 THEN 500
```

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.

The line ranges are sorted and overlapping ranges are merged before doing the DELETE.

Example

If lines exist in both ranges,

```
DELETE 20 TO 50, 10 TO 30
```

is the same as

```
DELETE 10 TO 50
```

If a line does not exist for a specified single line number, or there are no lines within a specified range, a message is displayed and the DELETE command is ignored.

The current line is set to the next line after the highest line number deleted. If such a line does not exist, the current line is set to the last line in the workspace.

Example

```
DELETE FIRST TO 130
```

deletes all lines up to and including 130.

Example

```
DELETE 100, 200 TO LAST
```

deletes line 100 and all lines after 199.

DROP Command

The DROP command erases the specified identifiers.

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 and arrays, without resetting the entire workspace. This will also free the *identifier* so that it may be reused. If no *identifiers* are specified in the DROP command, all currently active variables and arrays are forgotten.

Dropping a program *identifier* 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.

Variables and arrays of a compiled program unit cannot be DROPped. In this case, all variables and arrays referenced are treated as immediate variables and arrays, not as program variables and arrays. See "Immediate Statements" on page 133.

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 line ranges are sorted and overlapping ranges are merged before doing the EXTRACT.

If a line does not exist for a specified single line number or there are no lines within a specified range, a message is displayed and the EXTRACT command is rejected.

The current line is set to last program line in the workspace.

Example

If lines exist in both ranges,

```
EXTRACT 10 TO 30, 20 TO 50
```

is the same as

```
EXTRACT 10 TO 50
```

Example

```
EXTRACT 115 TO 120
```

Deletes all lines before 115 and all lines after 120 from the program, keeping lines 115 through 120.

Example

```
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.

FETCH Command

The FETCH command restores a program and its internal form to the workspace.

Format

```
FETCH program-name
Minimum:  FE
```

where:

program-name
is a file specification of the program to be fetched. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

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 with the program and its internal form (including any breakpoints in effect) as it was prior to the STORE. Unlike the LOAD command, which scans and translates to internal text, FETCH does not scan. Syntax errors are not reported, even if they exist. However, if you attempt to run a program and it contains errors, you are notified. You can use the LIST command with the ERROR clause to list any syntax errors.

Any program contained in the current workspace is cleared as part of FETCH. *Program-name* is used to derive the name of the workspace and *program-name* will be retained for a subsequent STORE command. Workspace naming rules are system dependent. See your IBM BASIC System Services manual for details.

If a *program-name* was being saved for a SAVE command, it is forgotten.

A successful FETCH sets the current line to the last line in the workspace.

If the specified file cannot be found or is invalid, an error message is displayed, leaving the workspace, the workspace name, and the current line setting unchanged.

FETCH can use only files created by the STORE command. The LOAD command, which works in conjunction with SAVE, should be used to load source program files.

Note: The FETCH command will recreate new internal format (as needed) for the program when it encounters a file generated by a prior release (a message is displayed if this occurs and, in this case, any syntax errors are reported). The file is not altered by the FETCH command. A STORE command should be issued to store the new format into the file.

Example

```
STORE PROGST
.
.
.
FETCH PROGST
.
.
.
```

FIND Command

The FIND command locates character strings within a program line or block of lines and displays the containing lines on the terminal.

Format

```
FIND [start-line [TO end-line]] delim string [delim [A[LL]]]
```

Minimum: FI

where:

start-line
may be a line number, or 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:

- *delim* is not contained in *string*.
- *delim* may be a space (or series of spaces) only if all the following conditions are true:
  - *string* starts with a letter (when *start-line* is omitted)
  - *string* is not FIRST, F, LAST, L (when *start-line* is omitted)
  - *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 (the case of each character must match). The string may not be null (may not have zero length).

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 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 the string are displayed.

If A[LL] is not specified, only the first line in the range containing the 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 message "STRING NOT FOUND" is displayed.

The leading line number and following space are not considered as part of the line to be searched.

Example

```
FIND /THE END/ALL
```

Find and display every line with the occurrence of the string "THE END."

Example

```
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

```
FIND /THE END/
```

Find and display the first occurrence of "THE END."

Example

```
FIND FOR
```

Find and display the first occurrence of "FOR."

Example

```
FIND VAR1 ALL
```

Find and display all occurrences of "VAR1."

GO Command

The GO command restarts a program that has been temporarily halted.

Format

```
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 when it has been suspended.

Execution is started with the RUN command.

Execution is suspended in one of the following 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
- Causing an ATTENTION interrupt (see your IBM BASIC System Services manual)
- The program is running in step mode

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 GOTO command 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/THEN/END IF blocks
- SELECT/CASE/END SELECT blocks

which could cause unpredictable results.

You may not specify a line number or use the GOTO command if execution is suspended in a compiled program unit.

GO END terminates the program and closes all files. You can use this version of the command independently of the current program unit. GO END may be specified while execution is suspended in a compiled program unit. GO END is equivalent to an immediate STOP statement.

If you modify the program in the workspace by using an editing command or entering a line of BASIC code, the execution of the program cannot be resumed. If no such change is made and neither an immediate STOP statement nor a GO END command has been executed, you can use the GO command to cause the program to continue. Commands which do not change the workspace are BREAK, DROP, HELP, LIST, PROFILE (as long as it executes only these commands), PURGE, QUERY, RENAME, SAVE, SET LOG, SET MSG, SET OPTION, SET PROFILE, SET TERM, STORE and SYSTEM.

Program execution may *not* be continued if a program variable or array is dropped (see "DROP Command" on page 438).

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 for each entry. If STEP is specified while execution is suspended in a compiled program unit, you will not step through statements in that program unit. But you will re-enter step mode when the next source statement in your workspace is executed or if the program chains to a source program. See "RUN Command" on page 472 for how to start a program in step mode.

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. GO without the STEP keyword resumes program execution; the step mode is cancelled.
- 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.

The keyword STEP may be specified with END, but has no effect on program execution. GO END STEP is the same as GO END.

Example

```
GO
```

continues execution.

Example

```
GO 500 STEP
```

continues execution in step mode at line 500.

Example

```
GOTO 200
```

continues execution at line 200.

Example

```
GO END
```

terminates program execution.

HELP Command

The HELP command displays information about BASIC.

Format

```
HELP [request]
Minimum:  HE
```

where:

request
is the name of a HELP panel. The name may be abbreviated.

Description

The HELP command has three primary uses:

1. as a quick reference you can invoke any time you want information about a BASIC statement, command, or function.
2. as a diagnostic aid to provide you with additional information about a BASIC message.
3. as a tutorial device you can use to learn about BASIC.

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 BASIC statements, intrinsic functions, commands, and diagnostics, as well as tutorial discussions of how to use the interactive facilities and how to write programs. Useful menu panels for main branches of the tree are:

BASIC (lists all non-message branches)

Statements (lists BASIC statements)

Functions (lists intrinsic functions)

Commands (lists BASIC commands)

Reswrd (lists reserved words)

Tutorial (describes the tutorial panels)

HELP Mode: HELP is a separate mode of the BASIC environment. Once you enter HELP mode (by executing a HELP command), you must use HELP *actions*. The usual commands, editing facilities, and immediate statements are not accepted until you leave HELP mode (by means of the "CAN" or "PRV" actions).

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, BASIC determines the panel to be displayed:

- If your entry of the HELP command was immediately preceded by a message, BASIC displays information concerning the message.
- If the HELP command was not preceded by a message, 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 referenced
- The actual current panel name (in parentheses)
- "PAGE n OF m"

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.

The bottom line displays the correspondence between PF keys and HELP actions. These PF keys are predefined and are *not* affected by the SET PF command.

Note: The syntax notation used in the HELP facility differs from that described in "Syntax Notation" on page 143. The HELP syntax notation is described in the HELP SYNTAX panel.

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 50 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 PF24	Returns control to BASIC command mode. Exits HELP mode (cancel).
HLP	PF1 PF13	Displays a panel which explains how to use HELP.

Figure 50 (Part 1 of 2). HELP Keywords and PF Keys

Keyword	PF Key	Action
PRV	PF3 PF15	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 BASIC command mode (same as CAN). Exits HELP mode.
SCF	PF8 PF20	Advances to the next page of the current panel (scroll forward). If there is no next page, a warning message appears on the command line.
SCB	PF7 PF19	Goes back to the previous page of the current panel (scroll backward). If there is no previous page, a warning message appears on the command line.
PRT	PF5 PF17	Prints all of the current panel to a file. See your IBM BASIC System Services manual for details. A message appears on the command line when the action is completed.

Figure 50 (Part 2 of 2). HELP Keywords and PF Keys

Printing All or Part of HELP: The PRT keyword may be used to print selected groups of HELP panels to a listing file. The following PRT requests are accepted:

```
PRT
PRT MESSAGES BAS
PRT MESSAGES BLI
PRT MESSAGES
PRT ALL
PRT panel name
```

When you enter only PRT (or PF5 or PF17), the current panel is printed.

When you specify MESSAGES BAS, all Processor messages are printed.

When you specify MESSAGES BLI, all Library messages are printed.

When you specify MESSAGES, all Processor messages and all Library messages are printed.

When you specify ALL, all HELP panels, including message panels, are printed in the following order:

general panels (how to use HELP, syntax, reserved words)
statements
functions
commands
tutorial
Processor messages (BAS)
Library messages (BLI)

When you specify a panel name, that panel is printed. If the panel is a menu panel, the panels listed in the menu are also printed. For example, if you say

```
PRT COMMANDS
```

The command menu panel and all of the command panels will be printed. Similarly, if you say

```
PRT BASIC
```

The topmost menu of the HELP tree will be printed, along with all the non-message panels.

Note: The PRT MESSAGES and PRT ALL commands generate a large file (there are over 1000 help panels available) and may take a long time to complete. An ATTN interrupt may be entered to terminate the command; the command is terminated when the panel being processed at the time of the interrupt has been printed. See your IBM BASIC System Services manual for how to cause an ATTN interrupt.

Note: The HELP listing file is system dependent. See your IBM BASIC System Services manual for details. If the listing file is a disk file, each PRT request re-writes the file; the previous contents are lost.

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   [program-name]
Minimum:  IN
```

where:

program-name
is a file specification to name the workspace. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

Description

The workspace is cleared of all statements and data.

If a *program-name* is specified, it is used to name the workspace; otherwise, the workspace is unnamed. Workspace naming rules are system dependent (see your IBM BASIC System Services manual for details).

If a *program-name* has been retained from a previous CHAIN statement, or a COMPILE, FETCH, INITIALIZE, LOAD, RENAME, or RUN command, it is deleted.

If a *program-name* is specified, it is retained for a subsequent SAVE command. The *program-name* will be forgotten if a COMPILE, FETCH, INITIALIZE, LOAD, RENAME, or RUN command, or a CHAIN statement causes another *program-name* to be retained.

Immediate options are reset to the defaults. (See "SET OPTION Command" on page 483 for default information.)

The current line setting is undefined.

Example

```
INITIALIZE MYPROG
```

The workspace is cleared and given the name MYPROG.

LIST Command

The LIST command is used to display some or all of the lines of the program in your workspace.

format

```
Format 1

    LIST [XREF] [ERROR[S] [ALL]] [OUT (list-file)]
        [line-number-range [,line-number-range]...]

Format 2

    LIST {FOR[WARD]|BACK[WARD]} [pages]

Minimum:  LI
```

where:

list-file
is a file specification of the listing file. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

line-number-range
specifies one or a range of line numbers in the program:

```
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].

pages
is an unsigned integer indicating the number of pages to be scrolled.

The XREF clause, ERROR clause, OUT clause, and line number ranges may appear in any order, but you may not specify the ERROR clause with the XREF clause. If the OUT clause appears last, you may omit the right parenthesis.

Format 1 Description

LIST causes the text of lines between the specified start and end-lines, inclusive, to be displayed. If you specify no line numbers, the entire workspace is displayed. If you specify a single (start) line, only that line is displayed.

While you may specify a listing of lines between nonexistent start and end lines, you will get an error message when you attempt the listing of a nonexistent individual line or when there are no lines within a specified range.

Lines appear in numeric order, and a line will be displayed only once, even if the line is included in more than one line number range. The line ranges are sorted and overlapping ranges are merged before doing the LIST.

Example

If there are lines within both ranges, then

```
LIST 20 TO 50, 10 TO 30
```

is the same as

```
LIST 10 TO 50
```

When a requested listing contains more lines than fit on the screen, the screen is filled and a message appears, asking you to press the ENTER key when you are ready to continue. If you respond with a null line (merely pressing the ENTER key), the listing continues to scroll until either the screen is full of new lines or the LIST command completes execution.

You may terminate your listing with an attention interrupt. See your IBM BASIC System Services manual for details about your system's interrupt procedures.

The LIST command sets the current line to the last line listed.

XREF Clause: Use the keyword XREF to obtain a cross-reference listing of the program. The listing is in three parts:

- Referenced line numbers in numeric sequence
- Line labels in alphabetic sequence
- 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.

ERROR Clause: The word ERRORS may be used in place of ERROR and has the same meaning.

If you specify ERROR without ALL, only the first line with syntax warnings or errors within the requested range is displayed together with its syntax messages. If you specify ERROR ALL, then all lines in the specified range(s) with syntax warnings or errors are displayed with their syntax messages. When more than one line is displayed, a separate informational message appears telling you the number of lines with errors. If there are no errors or warnings in the specified range(s), an informational message to this effect is displayed. Lines with FIPS violations are not displayed unless the line also has errors or warnings.

OUT Clause: Normally, listings are displayed at your terminal. However, you may use the OUT clause when you want to direct your output to a file. See your IBM BASIC System Services manual for valid list-file entries.

Example

```
LIST 110 TO 120
```

displays lines 110 through 120 of your program on the terminal.

Example

```
LIST 150 TO 250, 500 TO LAST
```

displays those lines in the range 150 to 250, and those from line 500 to the end of the program.

Example

```
LIST ERROR ALL OUT (MYFILE)
```

directs a listing of all errors in the program to the file named MYFILE.

Example

```
LIST XREF OUT (COSTFILE)
```

directs a cross–reference listing of the program in the workspace to the file named COSTFILE.

Format 2 Description

Format 2 of the LIST command includes one of the keywords: BACK[WARD] or FOR[WARD]. Your program will be scrolled backward or forward from the current line in *pages* of lines. If *pages* is omitted, 1 is assumed.

A page is the number of lines on the screen, minus 2 for the input area. The top line of a page is the bottom line of the preceding page. The bottom line of a page is the top line of the following page.

If a complete page is not on the terminal, a LIST BACKWARD will scroll backward one less page than specified in order to start the paging. For example, if only a portion of the page is on the terminal screen and you enter the LIST BACKWARD command, the page ending with the current line is displayed. Subsequent LIST BACKWARD commands will then scroll backward the specified number of pages.

If there is less than one page between the current line and the first line, LIST BACKWARD displays the first page of the program.

For both FORWARD and BACKWARD, if *pages* is 0, the current page is displayed or, if a complete page is not on the terminal, a page starting at the current line is displayed.

When scrolling forward, if a full page of lines does not fall on the specified page, the page is followed by the informational message:

```
* * *  END OF WORKSPACE  * * *
```

If the page is empty, the last line in the workspace is displayed before the message.

When scrolling backward, if a full page of lines is not in the specified page, the informational message:

```
* * *  TOP OF WORKSPACE  * * *
```

is displayed as the first line of the page, followed by as many lines as needed to make up the page. If there are not enough lines to fill the page, the informational message:

```
* * *  END OF WORKSPACE  * * *
```

is displayed after the lines.

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.

It is useful to set a PF key to the LIST command (see "SET PF Command" on page 485):

```
SET PF7 IMMED LIST BACKWARD
SET PF8 IMMED LIST FORWARD
```

Then PF7 can be used to scroll backward and PF8 to scroll forward a single page.

LOAD Command

The LOAD command clears the workspace and loads a program from your library.

Format

```
LOAD program-name
Minimum:  LO
```

where:

program-name
is a file specification of the program to be loaded. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

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.

The *program-name* is used to derive the name of the workspace. Workspace naming rules are system dependent (see your IBM BASIC System Services manual).

The *program-name* is retained for a subsequent SAVE command (see "SAVE Command" on page 477). A *program-name* previously retained for a SAVE command or a STORE command will be deleted.

The current line is set to the last line in the workspace.

If the specified file specification is not valid or the file cannot be found, an error message is displayed. The workspace, the workspace name, and the current line setting remain unchanged.

Lines in the file do not have to be ordered by line number; BASIC will automatically order them in the workspace. If the file contains more than one line with the same line number only the last line with that line number that is loaded will be in the workspace. (A message is displayed when a line with a duplicate line number is found.)

LOAD Command

LOAD is used in conjunction with the SAVE command. See "SAVE Command" on page 477.

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 a file into the program currently in the workspace.

Format

```
MERGE program-name [start-line] [TO end-line] [AT target-line]
    [STEP increment] [(REP[LACE] [)]]

Minimum:  ME
```

where:

program-name
is a file specification of the file to be merged. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

start-line *and* **end-line**
are line numbers that 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].

target-line
is a line number which is assigned to the first merged line, if specified.

increment
is an unsigned, 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 may be renumbered if the AT or STEP clauses are used.

The lines in the file must be in order by line number. No two lines in the file may have the same line number.

The current line is set to the last line merged.

AT Clause: The AT clause specifies where the merged lines should go in the workspace. The *target-line* number is assigned to the first merged line. If the AT clause is omitted, the line number assigned is the same as 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 to be renumbered. The increment between any two successive merged lines is as specified by the STEP clause. 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 renumbered range 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 range of merged lines are deleted.

Example

```
MERGE FILEA 350 TO 400 AT 350 STEP 5 (REPLACE)
```

Copies lines within the range 350 to 400 from the file 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

If the file named 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 into the workspace, but does not renumber them. If the line numbers 100, 150 or 200 are currently in the workspace, the merge will not take place.

Example

```
MERGE FILEA 350 TO 400 AT 300
```

Insert lines 350 to 400 from the file FILEA into the workspace starting at line 300. The merged lines are renumbered from 300 on with the increment the same as in the file FILEA.

Example

If the contents of a file named 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 210: NAME$,ADDRESS$ EXIT KEY_ERRORS
210 FORM C18,C40
.
.
.
```

The command

```
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
.
.
.
```

PROFILE Command

The PROFILE command executes a profile during a session.

format

```
PROFILE  profile-name
Minimum:  PR
```

where:

profile-name
is a file specification which specifies the file to be used as a profile. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

Description

The PROFILE command reads the specified file and acts as if the lines were entered at the terminal. The lines of the profile may contain any commands, immediate statements, and BASIC lines.

Display of lines read, as well as Processor informational and warning messages, is controlled by the SET PROFILE command (see "SET PROFILE Command" on page 487 for additional information).

Command output (for example, lines displayed by the LIST command) are displayed. Requests for command responses (including those from AUTO command and Format 2 of the CHANGE command) are displayed and the responses must be entered by the user.

If a BASIC line in the profile ends with an ampersand, the continuation line is read from the profile (the continuation line in the profile should contain the leading ampersand). For the AUTO and CHANGE commands, the user is prompted and the user must supply the continuation lines.

Program and immediate statement output (including exception messages) is displayed. Program input is requested from the user at the terminal. If BASIC was invoked as a compiler, all input and output is done in line mode even if SET TERM SCROLL is specified.

Error messages are displayed according to the current settings of the FLAG option and the SET MSG command.

Informationals, warnings, errors, and severe errors which occur while reading the profile do not cause termination of the reading of the profile. An unrecoverable error terminates the reading of all the profiles and displays a BASIC unrecoverable error message.

A QUIT command terminates reading of all profiles and BASIC returns to the host system.

A PROFILE command may be executed from within a profile. After the invoked profile completes execution, the invoking profile resumes. A profile may not cause a profile which is executing to be read again using the PROFILE command.

When designing a profile, care should be taken so that commands requiring responses are preceded by displayed information explaining the desired response (for example, by using immediate PRINT statements).

Example of a profile in file MYPROF

```
PRINT "HERE IS MY PROFILE"
SET PROFILE TERM
SET OPTION BASE 1, ARITHMETIC NATIVE
SET PF10 RETRIEVE
SET PF3 QUIT
SET LOG ON OUT(MYLOG)
```

To execute MYPROF profile, enter:

```
PROFILE MYPROF
```

PURGE Command

The PURGE command removes files from your library.

Format

```
PURGE file-name
Minimum:  PU
```

where:

file-name
is a file specification of the files to be purged. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

Description

The PURGE command deletes files associated with the *file-name* from your library. This may be more than one file. See your IBM BASIC System Services manual for a description of which files will be deleted for a given file-specification.

Example

```
PURGE MYFILE
```

Removes the files associated with MYFILE from your program library and makes the space they occupy available for use by other files.

QUERY Command

The QUERY command provides information about files.

Format

```
QUERY [file-type] [file-name] [OUT (list-file [)]]
Minimum:  QUE
```

where:

file-type
is one of the following keywords:

```
ALL
FILE[S]
PROG[RAM[S]]
WORK[SPACE[S]]
```

file-name *and* **list-file**
are file specifications. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

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 specified for *file-type*.

FILE means data files created by or for your programs.

PROGRAM means source program files, those created by the SAVE command.

WORKSPACE means files created by the STORE command.

File-name: If you specify a *file-name*, information regarding the requested types of files having that particular specification is displayed. If no *file-name* is stated, all files of the requested *file-type* are displayed.

OUT Clause: Normally, the requested data is displayed at the terminal, but it may be directed to *list-file* by using the OUT clause.

See your IBM BASIC System Services manual for detailed information on use of the QUERY command and system restrictions.

Example

```
QUERY PROGRAM
```

Obtains information about all source program files in your library.

Example

```
QUERY TEST
```

Obtains information about all files of all types with a *file-name* of TEST in your library.

Example

```
QUERY WORK
```

Obtains information about all the files in your library created by a STORE command.

Example

```
QUERY
```

Obtains information about *all* files of all types in your library, not just those created by or for BASIC. Use this judiciously; it could cause a very lengthy display if you have many files. (An attention may be entered to terminate the command if desired; see your IBM BASIC System Services manual.)

QUIT Command

The QUIT command ends the current BASIC session.

Format

```
QUIT [return-code]
Minimum:  QUI
```

where:

return-code
is an optionally signed integer constant. If not specified it defaults to zero.

Description

The QUIT command closes all open files, clears the workspace of all programs and data, and leaves the BASIC environment.

If you need to save the program in the workspace, execute a SAVE or STORE command before the QUIT command.

The *return-code* is passed to the host system. The *return-code* will be overridden if a nonzero IBM BASIC code is pending (for example, when an internal error has occurred and BASIC prompts you to SAVE or QUIT). See your IBM BASIC System Services manual for more details.

An error occurs if *return-code* is less than -2147483648 or greater than +2147483647 (in addition, the host system may impose further restrictions).

Example

```
SAVE MYPROG
QUIT 1234
```

or

```
STORE MYPROG
QUIT -999
```

BASIC will SAVE or STORE MYPROG and exit with a return code of 1234 or -999.

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 [program-name]
```

Minimum: RENA

where:

program-name
is a file specification used to name the workspace. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

Description

Program-name is used to rename the workspace. If *program-name* is not specified, you will be prompted for it. If you enter a null line in response to the prompt, the current name (if any) is unchanged.

The following rules apply:

- If *program-name* is not specified in the RENAME command, the name of the workspace is displayed.
- If *program-name* is not specified in the RENAME command, and a program name is being retained for a SAVE or STORE command, that program name is displayed.
- If a program name has been retained from a previous CHAIN statement, or COMPILE, FETCH, INITIALIZE, LOAD, RENAME, or RUN command, it is discarded if the workspace name is changed.
- If a *program-name* is entered (as part of the command or in response to the prompt), it is used to derive the workspace name. Workspace naming rules are system dependent (see your IBM BASIC System Services manual). The *program-name* is retained for a subsequent SAVE command.

If you enter the command:

```
RENAME NEWFILE
```

the current workspace name (if any) will be changed to NEWFILE and you will receive the message:

```
'NEWFILE' IS THE WORKSPACE NAME.
```

If you issue the command:

```
RENAME
```

and if the workspace has the name OLDFILE, and there is a file-specification being remembered for a SAVE, you will receive the prompt:

```
'OLDFILE' IS THE WORKSPACE NAME.
'program-name' REMEMBERED AS DEFAULT 'SAVE' NAME.
ENTER NEW NAME (NULL LINE FOR NO CHANGE): _
```

where **program-name** is the file specification being remembered for a SAVE and is system dependent (see your IBM BASIC System Services manual).

If the workspace is unnamed, you will receive the prompt:

```
THE WORKSPACE DOES NOT HAVE A NAME.
ENTER NEW NAME (NULL LINE FOR NO CHANGE): _
```

Entering a *program-name* (NEWPROG, for example) in response to either of these prompts will change the workspace name. You will receive the message:

```
'NEWPROG' IS THE WORKSPACE NAME.
```

If you entered nothing to the above prompts, BASIC will simply return to the ready mode (no action will be taken).

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 new-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*.

new-number
is a line number. The IBM-distributed default is 100.

increment
is an unsigned, nonzero integer less than 9999999. The IBM-distributed default increment is 10.

The order of 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 Clause: Line numbers are assigned to the program lines within the specified range by assigning *new-number* to the first program line within the range.

STEP Clause: 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 in the program (whether or not they are in the specified range) are changed, if necessary, to agree with the new line numbers.

The current line setting is unchanged; however, it may have a new line number associated with it.

Note: The use of line labels instead of line numbers for referencing lines will make renumbering faster as well as providing better program documentation.

The RENUMBER command issues error messages and the renumber will not be performed 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

```
RENUMBER 180 TO LAST AT 120 STEP 5
```

Renumber the lines from 180 to the end, start with 120 and increment by 5.

Example

```
RENUMBER
```

Renumber the entire program currently in the workspace, start with 100 and increment by 10.

Numeric constants that have a value equal to a line number will not be altered by the RENUMBER command (that is, they are not recognized by BASIC as line numbers).

Example

If the following line is in the workspace

```
100 IF LINE=200 THEN RETRY
```

and a RENUMBER command changes line number 200 to be 210, the numeric constant 200 in line 100 will *not* be changed to 210. You must make this change yourself.

RUN Command

The RUN command starts the execution of a program, starting with the lowest numbered statement.

Format

Format 1

```
RUN [program-name] [([SOURCE] [SPREC|LPREC] [PARM "literal"])] [STEP]
```

Format 2

```
RUN object-name (OBJ[ECT] [PARM "literal"]) [STEP]
```

Minimum: RU

where:

program-name or object-name
is a file specification of the program to be run. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

literal
is a character string.

The character string is within quotation marks or apostrophes. If a delimiting quotation mark (or apostrophe) appears in the string, it must be entered as two contiguous quotation marks (or apostrophes).

If a right parenthesis would appear last in the command, it may be omitted. If the ending quotation mark or apostrophe for the PARM string would appear last in the command, it may also be omitted.

General Description

Format 1 of the RUN command is used to run a source program. Format 2 is used to run a compiled object program.

Format 1 Description

If *program-name* is not specified, the program in the workspace is executed.

If you specify *program-name*, RUN first loads the program into the workspace as if the command

```
LOAD program-name
```

had been entered and execution is started. Note that the prior contents of your workspace are lost. The program lines are checked for proper syntax as they are brought into the workspace and error messages are displayed.

Note: The workspace will be changed if the program executes a CHAIN statement, so you may wish to SAVE the workspace before entering a RUN command.

SOURCE Clause: SOURCE designates execution of a source program. If you don't include it, SOURCE is assumed.

SPREC and LPREC Clauses: These options determine the maximum number of digits to be displayed by the PRINT statement (without the USING clause) when printing unformatted numeric values. SPREC and LPREC also determine the default precision of data defined by REAL statements or by the ARITHMETIC NATIVE option.

See"Data: Constants, Variables, and Arrays" on page 21 for more information about REAL data.

SPREC specifies "short" precision: 6 digits for printing and real single.

LPREC specifies "long" precision: 12 digits for printing and real double.

If you do not specify SPREC or LPREC, the default value will be used. The IBM-distributed default value is SPREC. Check with your system administrator for the initial default value for your organization. The SET OPTION command may be used to change the default. An OPTION statement within a program unit will override the RUN option.

PARM Clause: The PARM clause lets you pass a character string to the BASIC program invoked. The *literal* is used to specify the character string. Leading and trailing spaces are removed from the character string. The program may use the function PARM$ to retrieve the value (see "PARM$ Parameter" on page 403 for details).

Example

```
RUN TEST  (PARM ' 312 ')
```

where the character string would be "312."

STEP Clause: The STEP clause specifies execution in the step mode. In the step mode, processing halts before executing each statement (including the first) and displays an informational message notifying you that step mode is in effect and the line number of the next statement to be executed. Thus, you are allowed to interact with BASIC 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 445 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 multi-statement 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
- The function executes a BREAK or STOP statement, or
- An attention interrupt with SYSTEM action occurs while executing a multi-statement function,

a break occurs in the function. Step mode at the level of the referencing statement is not terminated if the ENTER key is pressed and the break was caused by other than a BREAK or STOP statement. A BREAK statement causes step mode to be terminated. To resume step mode after a BREAK statement, you must enter a GO command with the STEP option; step mode then starts at the level of the function.

Stepping ignores CALLs to compiled subprograms and CHAINs to compiled programs. Stepping resumes when the next statement to be executed is in the workspace or the program chains to a source program.

The RUN command initializes the values of all variables, arrays, and ON condition actions. Numeric variables and array elements are set to zero. Character variables and array elements are set to the null string. ON condition actions are set to SYSTEM. All files are closed. Immediate variables are dropped. TRACE and DEBUG are turned off.

The STEP clause 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. See "GO Command" on page 445 for additional information.

Example

```
RUN PROGA
```

Loads the program named PROGA in the workspace and executes it. It is equivalent to:

```
LOAD PROGA
RUN
```

Example

```
RUN MYFILE (LPREC)
```

Loads the source code of the program named MYFILE into the workspace and executes it with long precision printing of unformatted numeric values (maximum of twelve decimal digits). Also, real numbers default to real double unless specified as real single with a REAL SINGLE or DEFSNG statement.

Example

```
RUN MYFILE (SOURCE LPREC)
```

is equivalent to the preceding example.

Example

```
RUN (SPREC) STEP
```

Initiates execution of the program in the workspace in the step mode with short precision printing of unformatted numeric values (maximum of six decimal digits). Also, real numbers default to real single unless specified as real double with a REAL DOUBLE or DEFDBL statement, or by the # typing character.

```
RUN TESTPROG STEP
```

loads the program named TESTPROG into the workspace and executes it in step mode.

Format 2 Description

Format 2 of the RUN command executes a program that has been previously compiled and exists as an object module in your library. The workspace is cleared, and the named object module is loaded and executed.

SPREC or LPREC may not be specified in Format 2 with the OBJECT clause. When running object code, SPREC and LPREC remain as when the program was compiled.

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. TRACE and DEBUG are set off.

After the object module has been executed, the object module is cleared from the workspace, and the workspace does not have a name. Open files are closed.

OBJECT Clause: OBJECT specifies that the named file is a previously compiled object program you want to run.

PARM Clause: The PARM clause lets you pass a character string to a BASIC program when the program is invoked using the RUN command. PARM is used to specify the character string:

```
RUN PROG (OBJECT PARM 'A B COPY')
```

where the character string would be 'A B COPY'. The program may use the function PARM$ to retrieve the value (see "PARM$ Parameter" on page 403 for details).

STEP Clause: The STEP clause specifies execution in the step mode.

Since you are not able to STEP through object code, step mode will not go into effect until the object program chains to a source program.

In step mode, processing halts before executing each source statement (including the first) and displays the line number. See "STEP Clause" on page 473 for more details on step mode.

Example

```
RUN MYFILE (OBJ)
```

Clears the workspace and executes the object module named MYFILE.

SAVE Command

The SAVE command copies the source program in the workspace to a file.

Format

```
SAVE [program-name] [(REP[LACE] [)]]
Minimum:  SA
```

where:

program-name
is a file specification. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

Description

The program lines in your workspace are written to a file in line number sequence. The file is determined as follows:

1. *program-name*, if specified in the SAVE command.
2. If a file specification has been retained from a previous COMPILE, INITIALIZE, LOAD, RENAME, or RUN command, or a CHAIN statement, that file specification is prompted. You may accept it by responding with the ENTER key, or supplying a different file specification before pressing ENTER. You may cancel the command by clearing the input area and then pressing ENTER.
3. If the workspace has a name associated with it (set by the COMPILE, FETCH, INITIALIZE, LOAD, RENAME, and RUN commands or by a CHAIN statement), that name is prompted. You may accept it by responding with the ENTER key, or supply a different file specification before pressing ENTER. You may cancel the command by clearing the input area and then pressing ENTER.
4. If the workspace does not have a name, you are requested to enter a file specification. You may supply a file specification or cancel the command by entering a null line.

If the file specification entered, either as part of the SAVE command or in response to a prompt, is the same as the retained file specification or as the current name of the workspace (see your IBM BASIC System Services manual for details), the program in the workspace is saved to the file whether the file already exists or not. The keyword REPLACE is unnecessary and is ignored if present.

If the file specification is not the same as the retained file specification or as the current name of the workspace and there is an existing file of the same name, then you must explicitly state that an existing file is to be replaced. This may be done either by the REPLACE keyword or, if REPLACE is not used, a YES response to a message asking if the file should be replaced. A NO response may be entered to cancel the command. The responses YES and NO may be abbreviated. If

REPLACE is specified and the file does not already exist, an informational message is displayed and the file is saved.

The workspace and the workspace name are *not* changed by a SAVE command.

If a file specification is retained from the last COMPILE, INITIALIZE, LOAD, RENAME, or RUN command, or CHAIN statement, it will be discarded if you:

- Enter a COMPILE command (with *program-name*), FETCH, INITIALIZE, LOAD, or RUN (with *program-name* or *object-name*) command.
- Change the workspace name using the RENAME command.
- Execute a CHAIN statement.

Files created by the SAVE command may be loaded into the workspace with a LOAD command, RUN command, or COMPILE command and may be merged into the workspace with a MERGE command.

Example

```
SAVE PROGA
```

Writes PROGA from the workspace to your library. If PROGA already exists and if PROGA is not the name of the workspace, a prompt is issued asking if you really want to replace the file.

Example

```
SAVE PROGB (REPLACE)
```

Replaces the old version of PROGB with the new one. An informational message is displayed if no old version of PROGB exists and PROGB is not the name of the workspace.

SET LOG Command

The SET LOG command activates or deactivates logging of the BASIC dialog with the terminal.

Format

```
SET LOG {[ON] [OUT (log-name)] [APPEND]|OFF}
Minimum:  SE LOG
```

where:

log-name
is a file specification identifying the log file. See "Specifying Files in Commands" on page 138 for file specifications and the default file if you do not specify OUT. See also your IBM BASIC System Services manual.

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 to a file in the order they occur. Lines modified by using the full-screen edit feature are logged in the log file and tagged with a "+" prefix.

You may specify the logging file by naming one in an OUT clause; otherwise, logging is to a default file.

If neither ON nor OFF is 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 statements is not logged.
- HELP mode is not logged. See "HELP Command" on page 448.
- Operating system commands are not logged. This includes output from operating system commands and all dialog while in the operating system subset mode. (See "SYSTEM Command" on page 491.)

SET LOG ON commands when logging is ON result in an error message. SET LOG OFF when logging is not ON also results in an error message.

Unless you specify APPEND, each SET LOG ON command with an OUT clause erases the current contents of the specified logging file before logging recommences.

Unless you specify APPEND, the default file (when OUT has not been specified) is erased by the *first* SET LOG ON command in a BASIC session, but not by subsequent SET LOG ON commands. Thus, you may SET LOG ON and OFF to the default file repeatedly without loss of previous output.

When you specify APPEND, the designated file or the default file (if there is no OUT clause) is **not** erased and subsequent logging is appended to the file.

Example

```
SET LOG OUT (MYFILE)
```

sets logging on to the file MYFILE.

SET MSG Command

The SET MSG command controls the content of messages displayed at the terminal.

Format

```
SET MSG ({I|W|E|S}) {ALL|TEXT}
Minimum: SE MSG
```

Description

The SET MSG command controls how messages are displayed. A complete message consists of two parts:

1. Code. 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. Refer to Appendix A, "Exception Codes" on page 493 for the correspondence of these characters to exception codes.

   The final character of the code indicates the severity level of the message:

   I informative
   W warning
   E error with corrective assumption
   S error with no corrective assumption
   U unrecoverable error such that BASIC cannot continue normally. (Note that U may not be specified in the SET MSG command.)

2. Text. This is a variable number of characters which explain the reason for the message.

When ALL is in effect, both the code 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 exception, 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 a code.

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

For U level messages, both the code and the text are always displayed.

SET OPTION Command

The SET OPTION command may be used to define the default values for options.

Format

```
SET OPTION  option [, option]...
Minimum: SE OPTION
```

where:

option

may be any of the valid options from the OPTION Statement except RD.

```
ARITHMETIC {DECIMAL|NATIVE}
BASE {0|1}
COLLATE {NATIVE|STANDARD}
INVP [ON|OFF]
{SPREC|LPREC}
PRTZO nn
FLAG ({I|W|E|S})
{FIPS|NOFIPS}
```

See "OPTION Statement" on page 312 for a full discussion of the above options.

Description

The *option* values specified in the SET OPTION command become the default values for the options. *Option* values specified for FLAG and FIPS | NOFIPS in the SET OPTION command also become the values for the corresponding immediate options.

If a program unit is compiled and that program unit does not explicitly state a value for an option and that option is not specified on the COMPILE command or in the compiler invocation, then the default value at the time the program unit is compiled is used for that option.

If a program is executed with the RUN command (and the SOURCE option) and a program unit of the program does not explicitly state a value for the option, and that option is not specified on the RUN command, then the default value at the time the RUN command is entered is used as the value of that option for that program unit.

Default option values remain in effect throughout the BASIC session (unless reset by another SET).

The IBM-distributed initial default values are:

ARITHMETIC DECIMAL
BASE 0
COLLATE NATIVE
SPREC
PRTZO 20
FLAG (I)
NOFIPS
INVP use the value of the library installation option when the program is executed.

Note that the installation default for the option INVP does not affect a program until it is actually executed. The INVP default depends on which particular library is used when the program is executed.

If the default value for INVP is set to ON or OFF in a SET OPTION command, the INVP option does affect the compilation and overrides the library installation option.

If the system value is to use the library installation option, then specifying INVP without ON or OFF in a SET OPTION command will set the default to be the reverse of the library installation option. Otherwise, INVP without ON or OFF in a SET OPTION command will set the default to be to use the library installation option.

Initially, the immediate option values are the same as the default values and may be altered by immediate OPTION statements. See "Immediate Execution" of "OPTION Statement" on page 312 for more details.

Immediate option values revert to the default values when an INITIALIZE command or some other action occurs as described for "Immediate Execution" of the "OPTION Statement" on page 312.

SET PF Command

The SET PF command establishes PF key definitions for command input.

Format

```
Format 1
    SET PFn [IMMED|DELAYED] characters
Format 2
    SET PFn RETRIEVE
Format 3
    SET PFn
Minimum: SE PFn
```

where:

n
is an integer with a value from 1 to 24.

Description

1. Format 1 sets the PF key to a string of *characters*. An exclamation mark comment is considered to be part of the string. If you specify the IMMED option, the *characters* are executed immediately when you press the PF key, but if you specify the DELAYED option, you are given the opportunity to change the command before you enter it to continue. If you specify neither the IMMED nor the DELAYED option, DELAYED is the default.

2. Format 2 associates the RETRIEVE action with the PF key. Once you define a retrieve PF key, input lines in response to the command prompt (*) will be saved on a 500 character stack. When you use the retrieve PF key, the last input line on the stack will be recalled. If you use the retrieve PF key again, the next to last item will be recalled, and so on, until the stack is used up. Then, if you press the retrieve key, BASIC begins again with the first item.

   The input line retrieved by a retrieve key is not executed immediately. As in the DELAYED option of Format 1, the cursor is positioned after the line, and you may modify it before you enter it to continue.

   More than one PF key can be set to RETRIEVE.

3. Format 3 associates no action with the PF key.

   Note: You can't have (!) comments in Format 3; they would be considered as Format 1.

Initially, there are no actions associated with any of the PF keys.

When you use a SET PF command, the new function is assigned to the PF key even if that PF key has already been defined.

The definitions assigned by SET PF are acted upon only in command mode, and then only when SET TERM SCROLL is in effect. They do not affect the settings of PF keys in HELP or in BASIC programs.

AUTO Mode: When you are in AUTO mode and one of your PF keys is set to RETRIEVE, both the line number and the BASIC statement are saved on the stack. If you use the RETRIEVE key while in AUTO mode, the first item on the stack replaces the line number that has been generated, and AUTO mode is terminated unless the line number of the line as entered is the same as the line number that was prompted.

Example

```
* SET PF3 RETRIEVE
* auto
* 100 A = 5
* 110 B = 10
* 120 <PF3>

* 110 B = 10
*
```

The PF3 key recalled the last statement entered. Because the number of that statement, 110, is different from the line number it replaced (120), AUTO mode was terminated. Note that if you had changed the line number of the retrieved line back to 120 before pressing ENTER, AUTO mode would not have been terminated.

SET PROFILE Command

The SET PROFILE command lets you control display of lines and messages while reading a profile.

Format

```
SET PROFILE {TERM|NOTERM}
Minimum: SE PROFILE
```

Description

You could use SET PROFILE to verify the correctness of a profile, to debug profiles with errors, and for demonstration purposes.

If you specify TERM, any profile lines read will be displayed, together with informational and warning messages produced by the Processor, just as if the profile lines were typed at the terminal. Processor messages have a BAS prefix in their code (see "SET MSG Command" on page 481). If logging is on, these messages will be logged.

Note: Informational and warning messages will be displayed according to the current settings of the FLAG option and the SET MSG command. In particular, the FLAG option may suppress the display of informational and warning messages even if SET PROFILE TERM is in effect.

If you specify NOTERM (or if NOTERM is in effect by default), any profile lines read are not displayed, and informational and warning messages produced by the Processor are suppressed. These lines and messages are not logged.

When BASIC is invoked, NOTERM is the default.

An error occurs if neither TERM nor NOTERM is specified after SET PROFILE.

See "PROFILE Command" on page 462 for additional information.

SET TERM Command

The SET TERM command controls terminal input and output.

Format

```
SET TERM {SCROLL|LINE}
Minimum: SE TERM
```

Description

If the option SCROLL is specified, the terminal interaction for commands and programs is done in scrolling mode on 327x type display terminals. For other terminals, such as typewriter terminals, the SCROLL option is not meaningful and the command is ignored.

If the option LINE is specified, the terminal interaction (for commands and programs) is done in the operating system's line-mode whether or not the terminal is a display device.

The HELP mode of BASIC and the INPUT FIELDS and PRINT FIELDS statements are not affected by the SET TERM command.

The default when the BASIC environment is invoked is an installation option (the IBM-distributed default is SCROLL).

The compiler always displays in LINE mode.

The default when a compiled program is executed outside the BASIC environment is SCROLL.

The SET TERM command may be executed from a program (see "CALL BASIC Statement" on page 151).

Note: Selecting the SCROLL option may cause your terminal response time to perceptibly increase (this depends on the hardware connection between your terminal and the processor). You may want to select the LINE option to decrease the response time.

Note: If SET TERM LINE has been specified, Format 2 of the CHANGE command is not available and undermarks for error messages may not be placed correctly. Also, LIST FORWARD and LIST BACKWARD may not display complete pages on one screen (that is, a page may overlap onto the next screen).

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 [program-name] [(REP[LACE] [])]
Minimum:  ST
```

where:

program-name
is a file specification. See "Specifying Files in Commands" on page 138 and your IBM BASIC System Services manual for allowed file specifications.

Description

The program and its internal representation (including any breakpoints in effect) are written to a file. The file is determined as follows:

1. *program-name*, if specified in the STORE command.
2. If a file specification has been retained from a previous FETCH command, that file specification is prompted. You may accept it by responding with the ENTER key, or supply a different file specification before pressing ENTER. You may cancel the command by clearing the input area and then pressing ENTER.
3. If the workspace has a name associated with it (set by a CHAIN statement, the COMPILE, FETCH, INITIALIZE, LOAD, RENAME, or RUN commands), that name is prompted. You may accept it by responding with the ENTER key, or supply a different file specification before pressing ENTER. You may cancel the command by clearing the input area and then pressing ENTER.
4. If the workspace does not have a name, you are requested to enter a file specification. You may supply a file specification or cancel the command by entering a null line.

If the file specification entered, either as part of the STORE command or in response to a prompt, is the same as the retained file specification or as the current name of the workspace, (see your IBM BASIC System Services manual for details), the workspace is stored into the file whether the file already exists or not. The keyword REPLACE is unnecessary and is ignored if present.

If the file specification is not the same as the retained file specification or as the current name of the workspace and there is an existing file of the same name, then you must explicitly state that an existing file is to be replaced. This may be done either by the REPLACE keyword or, if REPLACE is not used, a YES response to a message asking if the file should be replaced. A NO response may be entered to cancel the command. The responses YES and NO may be abbreviated. If

REPLACE is specified and the file does not already exist, an informational message is displayed and the file is stored.

The workspace and the workspace name are *not* changed by a STORE command.

If a file specification is retained from the last FETCH command, it will be discarded if you:

- Enter a COMPILE (with *program-name*), FETCH, INITIALIZE, LOAD, or RUN (with *program-name* or *object-name*) command.
- Change the workspace name using the RENAME command.
- Execute a CHAIN statement.

Files created by the STORE command may be fetched into the workspace using the FETCH command (see "FETCH Command" on page 441).

Example

```
STORE MYFILE
```

Writes the contents of the workspace to MYFILE. If MYFILE already exists and if MYFILE is not the name of the workspace, a prompt is issued asking if you really want to replace the file.

Example

```
STORE MYFILE (REPLACE)
```

Replaces the old version of MYFILE with the new one. An informational message is displayed if no old version of MYFILE exists and MYFILE is not the name of the workspace.

SYSTEM Command

The SYSTEM command executes an operating system command or enters the operating system subset mode. This command is system dependent. See your IBM BASIC System Services manual for allowed operating system commands, details, and restrictions.

Format

```
SYSTEM ["operating-system-command["]]
Minimum:  SY
```

where:

operating-system-command
is one of the allowed operating system commands for your system.

Apostrophes may be used instead of quotation marks. If a delimiting quotation mark (or apostrophe) appears in an *operating-system-command*, it must be entered as two contiguous quotation marks (or apostrophes).

Description

If *operating-system-command* is included, it must be one of the allowed operating system commands. This command allows you to execute a single operating system command from BASIC.

Note: Be careful when using these commands; some of them can adversely affect your BASIC session.

If the command is rejected by the operating system, an error message is displayed.

If just the keyword SYSTEM (or an abbreviation) is entered, the BASIC environment is temporarily suspended and the operating system subset environment is entered. The subset environment may not be available on some operating systems. See your IBM BASIC System Services manual for restrictions and for details on how to return to BASIC from the subset environment.

Example

In the VM environment, if you enter

```
SYSTEM "LIST * BASIC *"
```

the names of all files of the type BASIC will be displayed.

In the TSO environment, if you enter

```
SYSTEM "ALLOCATE DDNAME(BASHELP) SYSOUT(A)"
```

the file associated with the name BASHELP will be directed to SYSOUT class A.

Appendix A. Exception Codes

The following table lists 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 shown following the exception code and explanation.

If none is shown for the ON condition, the exception always causes a message to be printed. If the code is underlined, then execution continues; if not underlined, execution of the program terminates.

The exception codes correspond to the message code (which is displayed with error messages) as follows: The 8-character MSGID (message identifier) is followed by a 9th character indicating the severity level (refer to page 481). The MSGID is in the form BLIsnnnn,

where:

BLI
identifies the IBM BASIC Library

s
represents the first digit of the exception code, either 0 or 1, for positive exception codes. For negative exception codes, s is a minus sign (-).

nnnn
is a 4-character field corresponding to the last 4 digits of positive exception codes and negative exception codes less than 10000. For negative 5-digit exception codes, the first *n* is

A
for codes in the range 10000 to 10999.

B
for codes in the range 11000 to 11999.

C
for codes in the range 12000 to 12999.

For example, BLI01001 is the MSGID of the error message for overflow in evaluating a numeric constant—which corresponds to the code 1001 in the table below. BLI-1009 is the MSGID for exception code -1009; BLI-C001 is the MSGID for exception code -12001.

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 MAT 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
-1016	Underflow in evaluating numeric constant.	UFLOW	none
-1017	Underflow in evaluating numeric intrinsic function.	UFLOW	none
1051	Overflow in evaluating character expression.	SOFLOW	SOFLOW
1052	Overflow in evaluating character MAT 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	SOFLOW
-1057	Overflow in character datum for PRINT FIELDS.	SOFLOW	SOFLOW
-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	CONV
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 or UDIM.	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.	none	none
-4102	CHAIN/USE parameter mismatch.	none	none
-4103	Subprogram parameter mismatch.	ERROR•	none
-4104	Function parameter mismatch.	ERROR	none
-4105	Illegal CHAIN to subprogram.	none	none
-4106	Illegal CALL to main program.	none	none
-4107	Illegal use of subprogram as main program.	none	none

• none for a called BASIC program. ERROR in the calling program unit for a CALL BASIC, CLINK, FLINK, PLINK, COBOL, FORTRAN, PLI, GDDM, SQL, or SYSTEM.

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
-5005	Insufficient storage for I/O buffer.	ERROR	IOERR
-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.	none	none
-5009	Insufficient storage for COMMON data area.	none	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 is not a square matrix.	ERROR	none
-6006	Array argument of AIDX, DIDX, DOT, or SRCH is not a one-dimensional array.	ERROR	none
-6008	Attempt to invert a singular matrix.	ZDIV	none
-6009	Destination array too small for MAT assignment.	ERROR	none
-6010	INPUT (PRINT) FIELDS: Field definition array is not one-dimensional.	ERROR	IOERR
6101	Mismatched dimensions in character array expression.	ERROR	none

FILE USE ERRORS (7000)		ON	EXIT
7001	File reference number not in range 0 to 255.	ERROR	IOERR
7002	File reference number zero not allowed in OPEN, CLOSE, RESET, or SCRATCH.	ERROR	IOERR
7003	Nonzero file reference number in OPEN already active.	ERROR	IOERR
7004	Inactive file reference 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
-7008	INPUT (PRINT) FIELDS file reference number not zero.	ERROR	IOERR
-7009	File in OPEN is already active.	ERROR	IOERR
-7010	I/O statement referenced a file that has become unavailable since last reference.	ERROR	IOERR
7201	File pointer cannot be reset as specified.	ERROR	IOERR
7202	RECORD clause not allowed for nonrelative 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
-7206	The rounded integer value of the record position is less than 1.	ERROR	IOERR
7301	Scratch of file not allowed.	ERROR	IOERR
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 for REWRITE exceeds current record length of the record in the file.	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 statement not allowed for display file.	ERROR	IOERR
-7312	FORM statement not allowed for internal file.	ERROR	IOERR
-7313	File not opened for OUTIN.	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 file.	ERROR	IOERR
-7318	Open OUTPUT or write for file that is read only.	ERROR	IOERR
-7319	Character overflow on PRINT to IMAGE/FORM spec.	SOFLOW	SOFLOW
-7320	File cannot be found.	ERROR	IOERR
-7321	OUTIN not allowed for STREAM files.	ERROR	IOERR
-7322	End media or extents.	ERROR	EOF
-7323	Invalid file attributes.	ERROR	IOERR
-7324	Records must be fixed length for relative file.	ERROR	IOERR
-7325	Invalid reset or position end on relative file.	ERROR	NOREC
-7326	Invalid reset or position end on keyed file.	ERROR	NOKEY
-7327	DEVICE PRINTER or 3800 expected but not found.	ERROR	IOERR

FILE USE ERRORS (7000)		ON	EXIT
-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
-7336	Attempt to write relative record without a REC= clause.	ERROR	IOERR
-7337	Invalid form specification for PRINT statement.	CONV	CONV
-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	Printer files (DEVICE PRINTER and/or 3800) must be OUTPUT and DISPLAY files.	ERROR	IOERR
-7342	Illegal file operation for file reference number zero.	ERROR	IOERR
-7343	Invalid record length specified.	ERROR	IOERR
-7346	READ required before REREAD.	ERROR	IOERR
-7347	Invalid margin value.	ERROR	IOERR
-7348	READ required before REWRITE.	ERROR	IOERR
-7349	Invalid record in INTERNAL file.	ERROR	IOERR
-7350	Invalid input operation with file pointer set at end by WRITE or PRINT.	ERROR	IOERR
-7351	Right margin error.	ERROR	IOERR
-7353	Font value must be between 1 and 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
-7361	NEWPAGE found in I/O list for file that is not a printer file (DEVICE PRINTER and/or 3800).	ERROR	IOERR
-7362	NEWPAGE found in FORM for file that is not a printer file (DEVICE PRINTER and/or 3800).	ERROR	IOERR
-7401	PF KEY and last input ignored.	SKEY	none

INPUT/OUTPUT ERRORS (8000)		ON	EXIT
8001	(MAT) READ beyond end of data.	ERROR	EOF [1]
8002	Too few data items in input reply.	ERROR	IOERR
8003	Too many data items in input reply.	ERROR	IOERR
-8004	Input reply length must not be greater than 156. Last input ignored.	ERROR	IOERR

[1] may use IOERR instead of EOF in EXIT statement.

INPUT/OUTPUT ERRORS (8000)		ON	EXIT
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
-8016	Attempt to position beyond end of record.	CONV	CONV
-8017	READ with no data statements declared in program unit.	ERROR	IOERR
-8018	INPUT (PRINT) FIELDS has more I/O list items than field definitions.	ERROR	IOERR
8101	Nonnumeric datum for (MAT) READ of numeric item.	CONV	CONV
8103	Nonnumeric datum for (MAT) INPUT of numeric item.	CONV	CONV
8104	Noncharacter 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	CONV
-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
-8112	Character datum for (MAT) input from FORM numeric specification.	CONV	CONV
8201	Invalid FORM.	ERROR	IOERR
8202	No data conversion specification in FORM or IMAGE.	ERROR	IOERR
-8203	Missing comma in FORM.	ERROR	IOERR
-8204	IMAGE only allowed 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	IOERR
-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 256 on input.	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
-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 field.	ERROR	IOERR

INPUT/OUTPUT ERRORS (8000)		ON	EXIT
-8313	INPUT (PRINT) FIELDS: Invalid syntax for row number.	ERROR	IOERR
-8314	INPUT (PRINT) FIELDS: Row number must be between 1 and maximum allowed.	ERROR	IOERR
-8315	INPUT (PRINT) FIELDS: Invalid syntax for column number.	ERROR	IOERR
-8316	INPUT (PRINT) FIELDS: Column number must be between 1 and maximum allowed.	ERROR	IOERR
-8317	INPUT (PRINT) FIELDS: Invalid syntax for FORM specifier.		
-8318	INPUT (PRINT) FIELDS: Field length must be between 1 and 156.	ERROR ERROR	IOERR IOERR
-8319	INPUT (PRINT) FIELDS: Incorrect syntax for leading and/or trailing attributes.	ERROR	IOERR
-8321	PIC syntax error.		
-8322	Attempt to read beyond end of record	ERROR	IOERR
-8324	Form scale specification is greater than width specification.	ERROR	CONV
-8326	Numeric value will not fit in form specification.	ERROR	IOERR
-8327	Syntactically incorrect data for INPUT FIELDS.	CONV	CONV
-8328	Numeric value will not fit in IMAGE specification. Replaced with asterisks.	ERROR CONV	IOERR 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 'file-spec'.	ERROR	none
-10005	Recursive function reference.	ERROR	none
-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 to program 'file-spec'.	none	none
-10013	Unable to load CALLed program unit 'subprogram'.	ERROR	none
-10014	Unable to remove chaining program when chaining to 'file-spec'.	ERROR	none
-10015	Program unit cannot be used.	none	none

SYSTEM ERRORS (11000)		ON	EXIT
-11001	System error returned on INPUT.	ERROR	IOERR
-11002	System error returned on CALL SYSTEM.	ERROR	none

SYSTEM ERRORS (11000)		ON	EXIT
-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	none
-11015	System error returned on CALL GDDM.	ERROR	none
-11016	System error returned on CALL COBOL.	ERROR	none
-11017	System error returned on CALL FORTRAN.	ERROR	none
-11018	System error returned on CALL PLI.	ERROR	none
-11019	System error returned on CALL BASIC.	ERROR	none
-11020	System error returned on CALL SQL.	ERROR	none

SPECIAL (12000)		ON	EXIT
-12001	Attention Interrupt.	ATTN	none
-12002	Unrecognized exception generated by CAUSE statement.	ERROR	none
-12101	Internal error in BASIC at adr.	none	none
-12201	Library installation option invalid in record N.	none	none
-12202	Error reading Library installation options file.	none	none
-12203	Product installation option invalid in record N.	none	none
-12204	Error reading product options file.	none	none
-12301	Internal error in BASIC. Report code X at address Y (R in M) to system administrator.	none	none
-12302	Log error—terminal logging suspended.	none	none
-12303	Program cannot be found or unable to load program.	none	none
-12304	Unable to load BASIC Library.	none	none
-12305	No user program name specified.	none	none
-12306	Insufficient storage available.	none	none
-12307	Unable to dynamically load a BASIC Library module.	none	none
-12308	Invalid BASIC Library index.	none	none
-12309	Paging errors in BLISEG, segment not used.	none	none
-12310	Unable to load BLISEG, segment not used.	none	none
-12311	Master directory invalid.	none	none
-12312	Paging errors in BASSEG, segment not used.	none	none
-12313	Unable to load BASSEG, segment not used.	none	none
-12314	BLISEG master directory invalid, segment not used.	none	none
-12315	Unable to load BASIC Processor.	none	none

Appendix B. CALL SQL Error Codes

Warning Messages

1 Warning: Underflow converting BASIC decimal to SQL decimal.
2 Warning: The statement to be deactivated was already inactive.
100 Warning: "Record not found" or End-of-table reached.

User Error Messages

11 Value-list contains too few entries for the number of columns to be retrieved and/or the indexed parameter(s).
12 A value-list array has more than one dimension.
13 Value list arrays have unequal dimensions.
14 Value list for SELECT columns contains both array and scalar arguments.
15 Value list arrays are invalid for index variables (except with the VALUES form of INSERT).
20 Number of rows requested is larger than the size of the value-list arrays.
21 Negative row count cannot be used with the VALUES form of INSERT.
30 Overflow converting BASIC DECIMAL to SQL DECIMAL or FLOAT.
31 Invalid data conversion implied: character to numeric.
32 Invalid data conversion implied: numeric to character.
33 Data base data type not supported by BASIC.
40 Statement is empty.
41 Statement does not start with a valid operation keyword.
42 Statement has an unmatched quote.
43 Statement cannot be executed. There are already 20 active statements.
44 Statement does not allow index variables.
45 Statement has an index variable that is negative or is not an integer.
46 Statement has index variable(s) without corresponding value list elements.
47 Statement uses an index variable that is zero or greater than 256.
50 Index variables that refer to array arguments can be used only in the VALUES clause of an INSERT statement.
51 Index variables cannot refer to both scalar and array arguments in the same statement.
52 Invalid use of an index variable in an INSERT statement.
60 SETOPT is not followed by the keyword NULL.
61 SETOPT statement does not end with an integer.
62 SETOPT statement has no value specified.
63 SETOPT statement has a null value that is not an integer or is too high.
64 Incorrect use of a value representing NULL. BASIC's represented null can be used explicitly only in the VALUES clause of INSERT or the SET clause of UPDATE.

User Error Messages

65 A REAL SINGLE argument is ambiguous. Current SETOPT value in force is too high to permit intended nulls to be distinguished from non-nulls.
70 CONNECT statement has a missing, misplaced, or unrecognizable keyword.
71 Index variable for userid or password in CONNECT statement has incorrect length or data type.
72 CONNECT statement is not valid for DB2.
80 SELECT list arguments are inconsistent with the original arguments for this statement (changed from character data type to numeric or vice versa).
81 Wording of COMMIT or ROLLBACK statement is incorrect.
90 Data base system is not available. CALL SQL cannot be used. (SQLWARN6="S")
91 User requested CANCEL of data base request.
92 DB2 stopping abnormally. Link terminating.
93 DB2 operator stopping database subsystem.

CALL SQL System Error Messages

101 Data base module (xxxxxxxx) not found.
102 Error during DB2 call - Return Code: nnn Reason Code: X'"xxx"
103 CALL SQL statement rejected due to previous system or internal error.

Appendix C. 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 IBM BASIC this collating sequence and character set is specified with the COLLATE STANDARD option (see "OPTION Statement" on page 312).

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

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
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	/		Slash
48	3/0	0		Zero
49	3/1	1		One
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	=		Equal 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

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
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 slash
93	5/13	]		Right bracket
94	5/14	¬ or ^		Logical NOT or 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
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

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
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 IBM BASIC this is specified with the COLLATE NATIVE option (see "OPTION Statement" on page 312).

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

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
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
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
79	4/15	|		Logical OR
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 or circumflex accent
96	6/0	-		Minus sign
97	6/1	/		Slash
106	6/10		VLN	Vertical broken line
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

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
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
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 slash
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

Ordinal Position	Ord Code	Graphic	Mnemonic	Name
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 D. 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 File statement, and the PRINT TO statement is replaced by the PRINT File 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 the BASE 1 option 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 ¬ sign) exponentiation symbol.)

VS BASIC Data Set Migration

If the specific data layout is known, IBM BASIC can handle VS BASIC stream-oriented files as IBM BASIC native type files, using FORM specifications.

With some restrictions, IBM BASIC programs can read VS BASIC nonstream files either as IBM BASIC display type files or as IBM BASIC native type files.

Glossary

The terms in this glossary are defined in accordance with their meanings in IBM 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.

Argument. An expression appearing in parentheses following a function or subprogram name when either is invoked. The expression represents a value (or array) that the function or subprogram is to act upon.

Arithmetic character. See Numeric character

Arithmetic constant. See Numeric constant

Arithmetic data. See Numeric data

Arithmetic expression. See Numeric expression

Arithmetic operator. See Numeric operator

Arithmetic variable. See Numeric 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, numeric 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. A single data item in an array; its position is indicated by a subscripted array reference.

Array expression. A numeric 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 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, and so forth.

Assignment statement. A statement that assigns data to variables or arrays.

Assignment symbol. The symbol =, which is used in an assignment statement to assign values to variables or arrays.

(B) BASIC. A term applied as a name to elements 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 with two terms. The binary operators that can be used in numeric 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 need 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 function or expression or 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 preceded by an exclamation mark (!), or as individual lines that start with the keyword REM. May be included on all IBM BASIC statements and commands except the DATA and IMAGE statements.

Concatenate. To join two strings in the order specified, thus forming one string whose length is equal to the sum of the lengths of the two strings.

Concatenation operator. An operator used in a character expression to join two character expressions. The concatenation operator is an ampersand (&).

Constant. A value that never changes. Constants can be numeric or character.

Control specification. One of the specifications X, POS, SKIP, or PAGE used in the FORM statement to specify formatting of records in native or display type files, or to control print line formatting.

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. One of the specifications used in the FORM statement to specify formatting of character and numeric values.

Data item. A single unit of data: a constant, a variable, an array element, or a function reference.

Data table. The values contained in the DATA statements of a BASIC program. DATA statements are processed in line 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 item in a READ statement. Initially, the indicator refers to the first item in the table. It can be repositioned 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 elements 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 type 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 identifies 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.

Exception. A circumstance arising during program execution when the semantic rules of BASIC cannot be followed or some resource constraint is about to be exceeded. Certain exceptions may be handled by automatic recovery procedures. 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 is 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 a program (for example, dividing by zero, branching to a nonexistent line number, and so forth.) See Exception.

Explicit declaration. The use of a DIM or COMMON statement to specify the number of elements in an array, the number of dimensions in an array, or the length of a character variable. The use of DECIMAL, INTEGER, and REAL 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 numeric or character value. Expressions can be constants, variables, or functions appearing alone, or in combination with operators. Five forms of expressions are defined in IBM BASIC: numeric, character, relational, logical, and array.

*** 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 notation. The form used to express a number consisting of one or more digits and a decimal point and optionally preceded by a sign.

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 notation. The form used to express a number consisting of an integer or fixed-point constant (the mantissa) followed by the letter E, followed by an optionally signed 1-3 digit integer constant (the exponent).

Formatted data. Data that has been input or output in accordance with the IMAGE or FORM statement.

Function. A named expression or block of statements that computes a single value. See also Intrinsic function and User-defined function.

Function reference. The appearance of an Intrinsic function name or a user-defined 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 subprogram, or a program.

Implicit declaration. The specification of the number of elements in an array or the number of dimensions in an array, either by a reference to an element of an array or by context (without the array being explicitly specified in a DIM or COMMON 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 element references, array references, and array references with redimensioning.

Input/output. The transfer of data between an external medium (the terminal or a file) and a program.

Input/output statement. A 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 notation. The form used to express an integer, consisting of one or more digits, optionally preceded by a sign.

Internal type file. A file created by 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 IBM BASIC.

*** 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, and so forth.) See Function.

Involution. See Exponentiation.

I/O. See Input/output.

*** Key.** One or more characters, within a set of data, that contains information about the set, including its identification.

Keyed file. A nonstream 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.

Library. A group of files accessible to the user.

Line. An ordered sequence of characters which terminates with an end of line.

Line number. The number which prefaces an IBM BASIC statement. It can be up to seven digits in length (in the range 1 to 9999999).

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 values 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 finite results.

Main program. The first program unit to receive control when a 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.

Multi-statement function. A user-defined function that is defined with more than one statement.

MVS. Multiple Virtual Storage. Refers to the MVS operating system.

Native type file. A file created by a 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 numeric 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. A constant with a numeric value. Numeric constants can be integer, decimal, or real.

Numeric data. Data having a numeric value. Numeric data can be integer, decimal, or real.

Numeric expression. 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.

Numeric variable. The name of a numeric 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 or percent sign (which specifies the type).

Numeric operator. A symbol representing an operation to be performed upon numeric data. The numeric operators are:

+	addition and unary plus sign
-	subtraction and unary minus sign
*	multiplication
/	division
^ or ¬ or **	exponentiation or involution

Object file (module). A file of internally coded data records appropriate for input to a linkage editor or loader.

Operand. A constant, a variable, an array element 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 enclosed in parentheses in a DEF statement, or a simple variable or empty array declarator enclosed in parentheses in 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 or logical operator; it is used when evaluating a numeric expression. The order of priorities, from high to low, is: exponentiation, unary operations, multiplication and division, addition and subtraction, relational operators, NOT, AND, OR. 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 statements that can be executed by the computer to attain a specific result.

Programmer-defined function. See User-defined function.

Program segmentation statement. Statements that provide a means to pass arguments between separate programs or program units. The statements CALL, CHAIN, END SUB, SUB, SUBEXIT, and USE are program segmentation statements.

(B) Program Unit. A self-contained part of a 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.

Real data. Floating-point binary data stored in System/370 floating-point format.

Record. A collection of related data items treated as a unit.

Redimension. To change the number of dimensions or the number of elements in each dimension of a previously declared array.

Redimension specification. The assignment of a new dimension specification to an already existing array, via an array assignment statement, or a GET, INPUT, INPUT File, LINPUT, LINPUT File, READ, READ File, or REREAD File statement.

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 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 or data item 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. A numeric 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.

Simple variable. A scalar variable (but not an array element).

Single-statement function. A user-defined function that is defined in one statement (the DEF statement).

Slot. The space for a data record in a relative record data file.

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.

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 numeric expression (whose rounded integer value is greater than or equal to zero) used to refer to a particular element 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.

*** Truncating.** To delete or omit a leading or of a trailing portion of a string in accordance with specified criteria.

TSO. Time Sharing Option.

TSO/E. Time Sharing Option/Extended.

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 numeric 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 using the services of a computing system.

User-defined function. A function defined by the user in a single-statement or multi-statement 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 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, dollar sign, or percent sign (which specifies the type).

Vector. (1) * A quantity usually represented by an ordered set of numbers. (2) A collection of array data having a single dimension.

VM. Virtual Machine. Usually refers to the VM operating system.

VM/SP. Virtual Machine/System Product.

VM/SP CMS. Virtual Machine/System Product Conversational Monitor System.

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.

Index

Special Characters

A

B

C

D

E

F

G

J

K

L

M

N

Q

R

S

T

Reader's
Comment
Form

IBM BASIC
Application Programming:
Language Reference
GC26-4026-2

This manual is part of a library that serves as a reference source for system 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: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should 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.

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last 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.

GC26-4026-2

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. S370-40) Printed in U.S.A. GC26-4026-2

IBM BASIC
Language Reference

File Number S370-40

GC26-4026-02

Printed in U.S.A.