GC26-3986-1
File No. S370-25

Systems

VS FORTRAN Application Programming: Language Reference

Program Numbers 5748-FO3 (Compiler and Library)
5748-LM3 (Library Only)

Release 1.1

IBM

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

Second Edition (January 1982)

This is a major revision of, and makes obsolete, GC26-3986-0, and its technical newsletter, GN26-0830.

This edition applies to Release 1.1 of VS FORTRAN, Program Products 5748-F03 (Compiler and Library) and 5748-LM3 (Library Only), 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 preface. 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 periodically made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/370 and 4300 Processors Bibilography, GC20-0001, for the editions that are applicable and current.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

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 reader's 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 any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

PREFACE

This manual outlines the programming rules for VS FORTRAN 1978-level source language. It includes Full American National Standard FORTRAN (X3.9-1978) plus IBM extensions.

After a brief introduction, the following subjects are discussed:

1. The VS FORTRAN language
2. Data

   Constants
   Variables
   Array elements
   Character substrings

3. Expressions

   Arithmetic
   Character
   Relational
   Logical

4. Statements (in alphabetic order)
5. Appendix

   Source Language Flagger (Includes execution-time cautions)
   VS FORTRAN-Supplied Procedures
   IBM and ANS FORTRAN Features
   Extended Error Handling Subroutines
   EBCDIC and ASCII Codes

If this book is revised, a summary of amendments will be included with the technical newsletter or new edition. Changes will be highlighted.

INDUSTRY STANDARDS

The VS FORTRAN Compiler and Library program product is designed according to the specifications of the following industry standards, as understood and interpreted by IBM as of June, 1980:

1. American National Standard Programming Language FORTRAN, ANSI X3.9-1978 (also known as FORTRAN 77).

   Portions of this manual are copied from American National Standard Programming Language FORTRAN, ANSI X3.9-1978. This material is reproduced, with permission, from American National Standards Institute, Incorporated, 1430 Broadway, New York, New York 10018.

2. International Organization for Standardization ISO 1539-1980 Programming Languages—FORTRAN.
3. American Standard FORTRAN, X3.9-1966.
4. International Organization for Standardization ISO R 1539-1972 Programming Languages-FORTRAN.

Standards 1 and 2 above are technically equivalent. When this manual refers to the current standard, it is referring to standards 1 and 2.

Standards 3 and 4 above are technically equivalent. When this manual refers to the **old standard**, it is referring to standards 3 and 4.

Both the FORTRAN 77 and the FORTRAN 66 standard languages include IBM extensions. When this manual refers to **current FORTRAN**, it is referring to the FORTRAN 77 standard plus the IBM extensions that are valid with it. When this manual refers to **old FORTRAN**, it is referring to the FORTRAN 66 standard plus the IBM extensions valid with it.

IBM VS FORTRAN PUBLICATIONS

The VS FORTRAN publications are designed to help develop programs with a minimum of wasted effort. This book, VS FORTRAN Application Programming: Language Reference, describes the rules for coding VS FORTRAN programs when using the current FORTRAN.

A series of related publications contain detailed documentation on writing programs using these rules:

- VS FORTRAN Application Programming: Guide, SC26-3985, contains guidance information on designing, coding, debugging, testing, and executing VS FORTRAN programs written at the current FORTRAN language level.
- VS FORTRAN Application Programming: Library Reference, SC26-3989, contains detailed information about the execution-time library subroutines.
- VS FORTRAN Application Programming: System Services Reference Supplement, SC26-3988, contains FORTRAN-specific reference documentation.
- VS FORTRAN Application Programming: Source-Time Reference Summary, SX26-3731, is a pocket-sized reference card containing current FORTRAN syntax and brief descriptions of the compiler options.
- System/360 and System/370 FORTRAN IV Language, GC28-6515, contains the rules for writing VS FORTRAN programs using FORTRAN 66.
- IBM System/370 Reference Data, GX20-1850.

Figure 1 shows how these manuals can be used together.

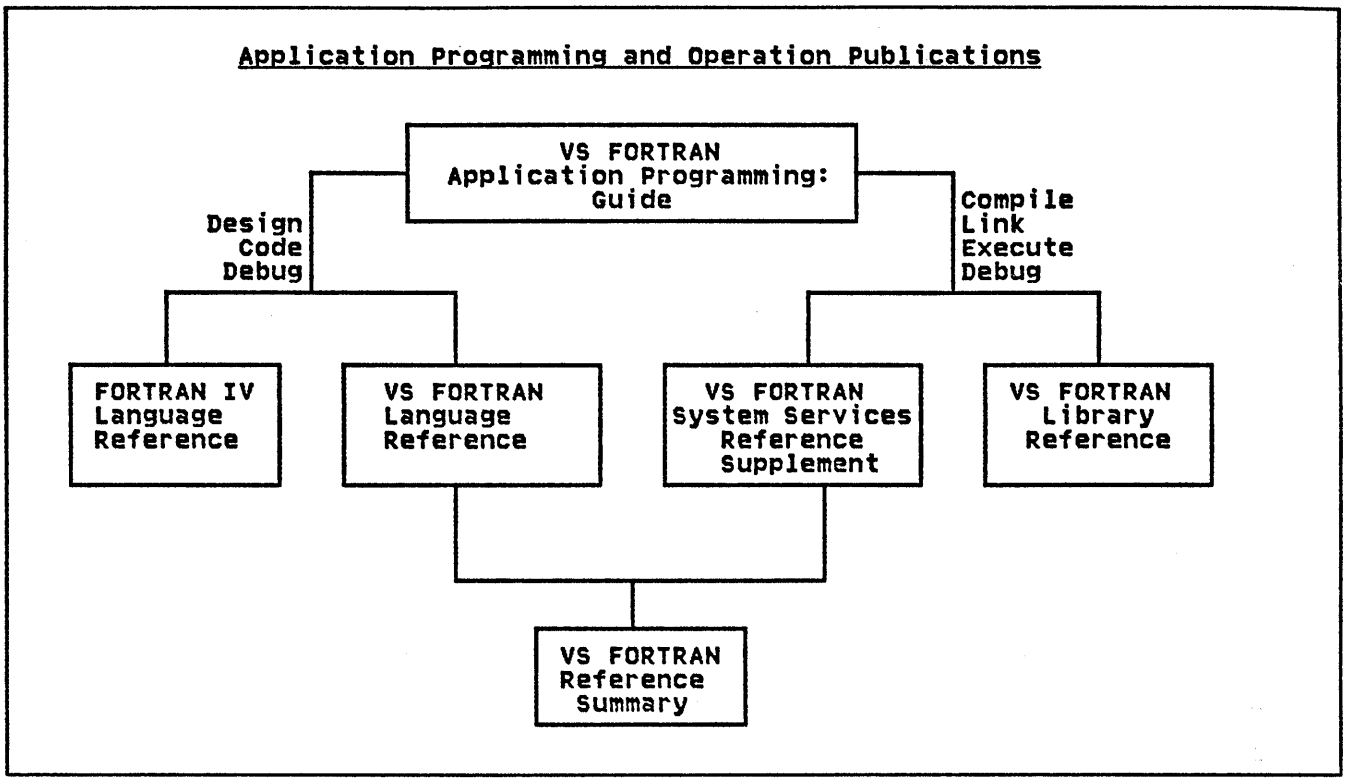

Figure 1. IBM VS FORTRAN Application Programming Publications

SUMMARY OF AMENDMENTS

RELEASE 1.1, JANUARY 1982

MISCELLANEOUS CHANGES

- Function subroutine charts have been added to Appendix B.
- The IBM extension abbreviations for .TRUE. and .FALSE. have been clarified.
- Several examples have been corrected.
- The syntax designations in the GO TO and logical IF statements have been corrected.
- Adjustments have been made to the ERRTRA subroutine description, including the Option Table Default Values.
- The index has been expanded.

RELEASE 1, JUNE 1981

MISCELLANEOUS CHANGES

- Unsigned arithmetic constants are explained.
- The IBM extension to the DATA statement is rewritten.
- The EJECT statement should not be continued.
- A logical IF statement containing *stn* may be labeled.
- The INCLUDE statement may not be continued.
- A parenthesis has been added to the PARAMETER statement syntax.
- MAIN# has been changed to MAIN in the PROGRAM statement.
- ID=*id* is a required parameter in the WAIT statement.

CONTENTS

FIGURES

INTRODUCTION

IBM VS FORTRAN consists of a language, a compiler, and an execution-time library of subprograms.

LANGUAGE

The VS FORTRAN language consists of a set of characters, conventions, and rules that are used to convey information to the compiler. The basis of the VS FORTRAN language is a statement containing combinations of element names, operators, constants, and words (keywords) whose meaning is predefined to the compiler.

The VS FORTRAN language is best suited to applications that involve mathematical computations and other manipulation of arithmetic data.

COMPILER

In a process called compilation, a program called the VS FORTRAN compiler analyzes the source program statements and translates them into a machine language program called the object program that can be combined with library routines to form a program suitable for execution. In addition, when the VS FORTRAN compiler detects errors in the source program, it produces appropriate diagnostic messages.

The VS FORTRAN compiler operates under control of an operating system that provides it with input, output, and other services. Object programs generated by the VS FORTRAN compiler also operate under operating system control and depend on it for similar services.

EXECUTION-TIME LIBRARY

The VS FORTRAN execution-time library consists of subroutines and functions supplied as part of the product. For complete information on the library, see VS FORTRAN Application Programming: Library Reference. For a brief description of the intrinsic functions and source subroutines to which the user may refer directly in VS FORTRAN statements, see "Appendix B. FORTRAN-Supplied Procedures" on page 204. For a discussion of extended error handling subroutines, see "Appendix D. Extended Error Handling Subroutines" on page 215.

Subroutines and functions to furnish any commonly used code sequences can be compiled and added to an execution-time library by the user. When written in VS FORTRAN, these can be structured as function, subroutine, or block data subprograms. Other source languages can be used if the subroutines are accessible by VS FORTRAN calls. User subroutines may reside directly in the supplied library data set or in a private data set called at load or link-edit time.

METHODS OF PRESENTATION

Because methods of presentation vary from book to book, the format notation and method of indicating IBM extensions are outlined here.

FORMAT NOTATION

In this manual, "must" is to be interpreted as a requirement; conversely, "must not" is to be interpreted as a prohibition.

In describing the form of VS FORTRAN statements or constructs, the following conventions and symbols are used:

- Special characters from the VS FORTRAN character set, uppercase letters, and uppercase words are to be written as shown, except where otherwise noted.
- Lowercase letters and lowercase words indicate general entities for which specific entities must be substituted in actual statements. Once a given lowercase letter or word is used in a syntactic specification to represent an entity, all subsequent occurrences of that letter or word represent the same entity until that letter or word is used in a subsequent syntactic specification to represent a different entity.
- Square brackets ([ ]) are used to indicate optional items.
- An underlined word (such as name, type, list) indicates a variable, such as an entry point, name of a function, data type, or list of variables or array names.
- An ellipsis (...) indicates that the preceding optional items may appear one or more times in succession.
- Blanks are used to improve readability; however, unless otherwise noted, they have no significance.

The general form of each statement is enclosed in a box. For example:

Syntax

```
CALL name [ ( [arg1 [,arg2] [,arg3] ... ] ) ]
```

The following examples are among those allowed:

```
CALL name
CALL name ()
CALL name (arg)
CALL name (arg, arg)
CALL name (arg, arg, arg)
CALL name (arg, arg, arg, arg)
```

When an actual statement is written, specific entities are substituted for name and each arg. For example:

```
CALL ABCD (X,1.0)
```

DOCUMENTATION OF IBM EXTENSIONS

In addition to the statements available in FORTRAN 77, IBM provides "extensions" to the language. These extensions are shown in the following ways.

IBM EXTENSION

This paragraph shows how IBM language extensions in text are documented.

END OF IBM EXTENSION

The following example shows how boxes indicate IBM extensions.

Name	Type	Length
I, J, K	Integer variables	4 [, 2, 2]
C	Real variable	4
D	Complex variable	[16]

The example below shows how IBM extensions are documented within a table. The boxes around certain types and lengths of the result of logical operations indicate IBM extensions.

First Operand / Second Operand	[Logical (1)]	Logical (4)
[Logical (1)]	[Logical (4)]	[Logical (4)]
Logical (4)	[Logical (4)]	Logical (4)

VALID AND INVALID VS FORTRAN PROGRAMS

This manual defines the rules (that is, the syntax, semantics, and restrictions) applicable for writing valid VS FORTRAN programs either for the 1978 Standard or for the 1978 Standard plus IBM extensions. Most violations of the VS FORTRAN language rules are diagnosed by the compiler; however, some syntactic and semantic combinations are not diagnosed, some because they are detectable only at execution time, others for performance reasons. VS FORTRAN programs that contain these undiagnosed combinations are invalid VS FORTRAN programs, whether or not they execute as expected.

VS FORTRAN LANGUAGE

A VS FORTRAN program is made up of three basic elements:

Data Consists of constants, variables, and arrays. See "VS FORTRAN Data" on page 11.

Expressions Executable sets of arithmetic, character, logical, or relational data. See "VS FORTRAN Expressions" on page 25.

Statements Combinations of data and expressions. See "VS FORTRAN Statement Descriptions" on page 45.

LANGUAGE DEFINITIONS

Some of the terms used in the discussion of the VS FORTRAN programming language are defined as follows:

Main program. A program unit, required for execution, that can call other program units but cannot be called by them. A main program does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. The main program is the first to receive control at execution time.

Subprogram. A program unit that is invoked by another program unit in the same program. In FORTRAN, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

Procedure. A sequenced set of statements that may be used at one or more points in one or more computer programs, and that usually is given one or more input parameters and returns one or more output parameters. A procedure consists of subroutines, function subprograms, and intrinsic functions.

Intrinsic function. A function, supplied by VS FORTRAN, that performs mathematical or character operations. (See "INTRINSIC Statement" on page 130.)

External procedure. A subroutine or function subprogram written in FORTRAN.

Executable program. A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures, or both.

Executable statement. A statement that calculates, tests, or alters the flow of control.

Nonexecutable statement. A statement that describes the characteristics of the program unit, of data, of editing information, or of statement functions, but does not cause an action to be taken by the program.

Preconnected file. A unit or file that was defined at installation time. However, a preconnected file does not exist for a program if the file is not defined by a FILEDEF command or by job control statements.

Program unit. A sequence of statements constituting a main program or subprogram.

Additional definitions can be found in the "Glossary" on page 241.

LANGUAGE SYNTAX

For the compiler to understand instructions, certain syntax rules must be carefully adhered to when entering the following items. Each of these items is discussed more fully following the list.

Source language statements
Source statement characters
Names
Statement numbers
Keywords

SOURCE LANGUAGE STATEMENTS

VS FORTRAN accepts source input in either of two formats:

- Fixed-form input format: Fixed-length 80-byte records.

IBM EXTENSION

- Free-form input format: Fixed-length records (with a maximum length of 1320 bytes). This frees the programmer from card column restrictions and is useful for terminal input.

END OF IBM EXTENSION

A program unit must be written in either fixed form or free form—not both.

Fixed-Form Source Statements

The statements of a VS FORTRAN source program can be written on a standard FORTRAN Coding Form, GX28-7327. Each line on the coding form is 80 characters long and is equivalent to one 80-column card (or input line on a terminal).

- Statement number

  The statement number consists of from 1 to 5 decimal digits. It must not be zero. Blanks and leading zeros in a statement number are ignored. The values of the statement numbers do not affect the order in which the statements are executed. The same statement number must not be given to more than one statement in a program unit.

- Initial line

  VS FORTRAN statements are written within columns 7 through 72. The first line of a statement may have a number in columns 1 through 5 and must have a blank or zero in column 6.

- Comments

  Comments to explain the program may be written in columns 2 through 72 if the letter C or an asterisk (*) is placed in column 1. The VS FORTRAN compiler does not process comments other than to print them as part of the source program listing. Comments may appear anywhere in the program before the END statement. Blank lines can appear anywhere in the program and are processed as comments.

- Continuation Line

  A VS FORTRAN statement that cannot be completed on one line may be continued onto as many as 19 additional lines. A continuation line has any character other than a blank or zero in column 6. The statement is then continued within columns 7 through 72.

  Comments can appear between continuation lines.

IBM EXTENSION

VS FORTRAN allows columns 1 through 5 in a continuation line to contain any characters, but they are ignored.

END OF IBM EXTENSION

- Identification

  Columns 73 through 80 of any VS FORTRAN line are not significant to the compiler and may, therefore, be used for identification, sequencing, or any other purpose.

As many blanks as desired may be written in a statement or comment to improve its readability. They are ignored by the compiler. However, blanks that are inserted in literal or character data are retained and treated as blanks within the data.

Figure 2 illustrates fixed-form source statements.

```
Column:  1    67
         ----------------------
         C     SAMPLE TEXT
                  .
                  .
                  .
           10 D=010.5
              GO TO 56
          150 A=B+C*(D+E**F+
              1G+H-2.*(G+P))
              C=3.
                  .
                  .
                  .
```

Figure 2. Example of Fixed-Form Source Statements

IBM EXTENSION

Free-Form Source Statements

The following rules govern free-form input format (free-form source):

- Statement number

  The initial line may contain, as the first nonblank character of that line, a statement number consisting of from one to five decimal digits. Blanks and leading zeros in a statement number are ignored. A blank need not separate a statement number from the first nonblank character that follows the statement number.

- Initial line

  An initial line is the first line of the statement and may start in any position on a new line.

- Comments

  A comment line is a line that does not follow a continued line and that has a quotation mark (") in the first character position (column 1). Blank lines are not allowed as comment lines. A comment line cannot be continued.

- Continued line

  A line of a statement to be continued is indicated by terminating the line with a hyphen or minus sign (-). A comment line cannot be continued.

- Preserving a minus sign

  If the last character in the line is a hyphen (minus sign), it is assumed to indicate continuation and is discarded. If the last two characters in a line are hyphens, only the last one is taken as a continuation character; the preceding one is preserved as a minus sign.

- Continuation line

  A continuation line is a line following a continued line. It may start in any position. Up to 19 continuation lines are permitted in a single statement.

- Maximum statement length

  The maximum length of a free-form source statement is 1320 characters, excluding the statement continuation character and the statement number. Blank characters are counted in the total number of characters.

Figure 3 illustrates free-form source statements.

```
Column:   1     7
          --------------------
          "SAMPLE TEXT
            .
            .
            .
          10D=010.5
          GO TO 56
          150 A=B+C*(D+E**F+-
          G+H-2.*(G+P))
          C=3.
            .
            .
            .
```

Figure 3. Example of Free-Form Source Statements

END OF IBM EXTENSION

SOURCE STATEMENT CHARACTERS

The characters listed in Figure 4 on page 8 constitute the set of characters acceptable in a VS FORTRAN program.

A special character may be an operator (or part of an operator), part of a constant, or have some other special meaning. The interpretation is implied by the context.

The special characters shown in Figure 4 on page 8 are listed in their correct collating sequence. (The complete collating sequence can be found in "Appendix E. EBCDIC and ASCII Codes" on page 236.)

Special Characters		Letters		Digits
	blank	A	O	0
.	period	B	P	1
(	left parenthesis	C	Q	2
+	plus sign	D	R	3
$	currency sign	E	S	4
*	asterisk	F	T	5
)	right parenthesis	G	U	6
-	minus sign	H	V	7
/	slash	I	W	8
,	comma	J	X	9
:	colon	K	Y	
'	apostrophe	L	Z	
=	equal sign	M		
"	quotation mark	N	$	

Figure 4. Source Statement Characters

NAMES

Names (referred to as "symbolic names" in old FORTRAN publications) can be assigned to the elements of a program unit.

Definition

Name—A string of 1 through 6 letters (A,B,...,Z) or digits (0,1,...,9), the first of which must be a letter.

IBM EXTENSION

With this compiler, the currency symbol ($) is treated as a letter when used in a name. Therefore, the currency symbol ($) can be used as the first character in a name.

END OF IBM EXTENSION

Names can be used to identify the following items in a program unit:

- An array and the elements of that array (see "Array Elements" on page 20)
- A variable (see "Variables" on page 18)
- A constant (See "PARAMETER Statement" on page 138)
- A main program (see "PROGRAM Statement" on page 141)
- A statement function (see "Statement Function Statement" on page 169)
- An intrinsic function (see "Appendix B. FORTRAN-Supplied Procedures" on page 204)
- A function subprogram (see "FUNCTION Statement" on page 111)
- A subroutine subprogram (see "SUBROUTINE Statement" on page 173)
- A block data subprogram (see "BLOCK DATA Statement" on page 56)

- A common-block (see "COMMON Statement" on page 62)
- An external user-supplied subprogram that cannot be classified by its usage in that program unit as either a subroutine or function subprogram name (see "EXTERNAL Statement" on page 89)
- A NAMELIST (see "READ Statement with NAMELIST" on page 162 and "WRITE Statement with NAMELIST" on page 198)

A name that identifies a constant, variable, array, external function, or statement function also identifies its data type. The name may be specified in a specification statement (see "Specification Statements" on page 43). If the name does not appear in such a statement, the type is implied by the first letter of the name. A first letter of I through N implies integer type, and any other letter (or the currency symbol) implies real type, unless an IMPLICIT statement is used to change the default type.

Names are either global or local.

- Classes of global names:
  - Common block
  - External function
  - Subroutine
  - Main program
  - Block data subprogram
- Classes of local names:
  - Array
  - Variable
  - Constant
  - Statement function
  - Intrinsic function
  - Dummy procedure

Names must be unique within a class in a program unit and can identify elements of only one class except in the following situations:

- A common-block name can also be an array, variable, or statement function name in a program unit.
- A function subprogram name must also be a variable name in the function subprogram.

The name of a main program, subroutine, common-block, NAMELIST, or block data subprogram has no type. A generic function name has no predetermined type; it assumes a type dependent upon the type of its argument(s).

Once a name is used as a main program name, a function subprogram name, a subroutine subprogram name, a block data subprogram name, a common-block name, or an external procedure name in any unit of an executable program, no other program unit of that executable program can use that name to identify an entity of these classes in any other way.

STATEMENT NUMBERS

Statement numbers identify statements in a VS FORTRAN program.

A statement number is a sequence of from one to five digits, one of which must be nonzero. It can be written in either fixed form or free form. See "Statement Numbers" on page 171.

KEYWORDS

Keywords identify VS FORTRAN-supplied procedures (intrinsic functions) that can be used as part of any program. These procedures are mathematical functions and service subroutines that are supplied to save programmers the time it would take to write them every time that particular sequence of statements is needed in a program. See "Appendix B. FORTRAN-Supplied Procedures" on page 204.

A keyword is a specified sequence of characters. Whether a particular sequence of characters identifies a keyword or a name is implied by context. There is no sequence of characters that is reserved in all contexts.

VS FORTRAN DATA

Data is a formal representation of facts, concepts, or instructions. VS FORTRAN manipulates three general kinds of data:

- Constants
- Variables
- Arrays

Note: These are not to be confused with data types. Data types correspond to the the five types of variables, as discussed under "Variable Types and Lengths" on page 18.

CONSTANTS

A constant is a fixed, unvarying quantity. There are several classes of constants:

- **Arithmetic** constants specify decimal values:

  Integer
  Real
  Complex

- **Logical** constants specify a logical value as "true" or "false." There are two logical constants:

  .TRUE.
  .FALSE.

- **Character** constants are a string of alphameric and/or special characters enclosed in apostrophes.
- **Hollerith** constants are used only in FORMAT statements.

IBM EXTENSION

- **Hexadecimal** constants are used only as data initialization values of arithmetic or logical variables.

END OF IBM EXTENSION

The PARAMETER statement allows a constant to be given a name. (See "PARAMETER Statement" on page 138.)

ARITHMETIC CONSTANTS

Arithmetic constants fall into three categories: integer, real, and complex.

An unsigned constant is a constant with no leading sign. A signed constant is a constant with a leading plus or minus sign. An optionally signed constant is a constant that may be either signed or unsigned. Only integer and real constants may be optionally signed.

Integer Constants

Definition

Integer Constant—A string of decimal digits containing no decimal point and expressing a whole number. It occupies 4 bytes of storage.

Maximum Magnitude: 2 147 483 647 (that is, $2^{31}-1$).

An integer constant may be positive, zero, or negative. If unsigned and nonzero, it is assumed to be positive. (A zero may be written with a preceding sign with no effect on the value.) Its magnitude must not be greater than the maximum and it must not contain embedded commas.

Valid Integer Constants:

```
0
91
173
-214 748 3647
```

Invalid Integer Constants:

27.	Contains a decimal point.
3145903612	Exceeds the maximum magnitude.
5,396	Contains an embedded comma.
-2147483648	Exceeds the maximum magnitude, even though it fits into 4 bytes.

Real Constants

Definition

Real Constant—A string of decimal digits that expresses a real number. It can have one of three forms: a basic real constant, a basic real constant followed by a real exponent, or an integer constant followed by a real exponent.

A basic real constant is a string of digits with a decimal point. It is used to approximate the value of the constant.

The storage requirement (length) of a real constant can also be explicitly specified by appending an exponent to a basic real constant or an integer constant. The standard exponents consist of the letters E and D.

IBM EXTENSION

This compiler also allows the letter Q as an exponent.

END OF IBM EXTENSION

An exponent is followed by a signed or unsigned 1- or 2-digit integer constant. The letter E specifies a constant of length 4; the letter D specifies a constant of length 8.

IBM EXTENSION

The letter Q specifies a constant of length 16.

END OF IBM EXTENSION

Magnitude: 0 or 16^{-65} (approximately 10^{-78}) through 16^{63} (approximately 10^{75})

Precision: (Four bytes) 6 hexadecimal digits (approximately 6 decimal digits)

(Eight bytes) 14 hexadecimal digits (approximately 15 decimal digits)

IBM EXTENSION

(Sixteen bytes) 28 hexadecimal digits (approximately 32 decimal digits)

END OF IBM EXTENSION

A real constant may be positive, zero, or negative (if unsigned and nonzero, it is assumed to be positive) and must be within the allowable range. It may not contain embedded commas. A zero may be written with a preceding sign with no effect on the value. The decimal exponent permits the expression of a real constant as the product of a basic real constant or integer constant and 10 raised to a desired power.

Valid Real Constants (Four Bytes):

+0.	
-999.9999	
7.0E+0	That is, 7.0 x 10^0 = 7.0
9761.25E+1	That is, 9761.25 x 10^1 = 97612.5
7.E3	
7.0E3	That is, 7.0 x 10^3 = 7000.0
7.0E+03	
7E-03	That is, 7.0 x 10^{-3} = 0.007
21.98753829457168	Note: This is a valid real constant, but it cannot be accommodated in four bytes. It will be accepted and truncated.

Valid Real Constants (Eight Bytes):

1234567890123456.D-73	Equivalent to .1234567890123456x10^{-57}
7.9D03	
7.9D+03	That is, 7.9 x 10^3 = 7900.0
7.9D+3	
7.9D0	That is, 7.9 x 10^0 = 7.9
7D03	That is, 7.0 x 10^3 = 7000.0

IBM EXTENSION

Valid Real Constants (Sixteen Bytes):

.2345234534564567345656780+43

5.001Q08

END OF IBM EXTENSION

Invalid Real Constants:

1	Missing a decimal point or a decimal exponent.
3,471.1	Embedded comma.
1.E	Missing a 1- or 2-digit integer constant following the E. It is not intepreted as 1.0 x 10^0.
1.2E+113	Too many digits in the exponent.
23.5D+97	Magnitude outside the allowable range, that is, 23.5 x 10^{97}>16^{63}.
21.3D-99	Magnitude outside the allowable range, that is, 21.3 x 10^{-99}<16^{-65}.

IBM EXTENSION

88.632157480123	Too many digits in the exponent

END OF IBM EXTENSION

Complex Constants

Definition

Complex Constant—An ordered pair of signed or unsigned integer or real constants separated by a comma and enclosed in parentheses. The first constant in a complex constant represents the real part of the complex number; the second represents the imaginary part of the complex number.

The real or integer constants in a complex constant may be positive, zero, or negative and must be within the allowable range. (If unsigned and nonzero, they are assumed to be positive.) A zero may be written with a preceding sign, with no effect on the value. If both constants are of type integer, however, then both are converted to type real of length 4 bytes.

IBM EXTENSION

If the constants of the ordered pair representing the complex constant differ in precision, the constant of lower precision is converted to a constant of the higher precision.

For example, if one constant is real and the other is double precision, real is converted to double precision.

If the constants differ in type, the integer constant is converted to a real constant of the same precision as the original real constant.

For example, if one constant is integer and the other is double precision, then the integer constant is converted to a double precision constant.

END OF IBM EXTENSION

Valid Complex Constants (i = square root of -1):

(3,-1.86)	Has the value 3.- 1.86i; both parts are real (4 bytes long).

IBM EXTENSION

(-5.0E+03,.16D+02)	Has the value -5000.+16.0i; both parts are double precision.
(4.7D+2,1.973614D4)	Has the value 470.+19736.14i.
(47D+2,38D+3)	Has the value 4700.+38000.i.
(1234.345456567678Q59,-1.0Q-5)	
(45Q6,6E45)	Both parts are real (16 bytes long.)

END OF IBM EXTENSION

Invalid Complex Constants:

(A, 3.7)	Real part is not a constant.

IBM EXTENSION

(.0009Q-1,7643.Q+1199)	Too many digits in the exponent of the imaginary part.
(49.76, .015D+92)	Magnitude of imaginary part is outside of allowable range.

END OF IBM EXTENSION

LOGICAL CONSTANTS

Definition

Logical Constant—A constant that can have a logical value of either true or false.

There are two logical constants:

.TRUE.
.FALSE.

The words TRUE and FALSE must be preceded and followed by periods. Each occupies 4 bytes.

IBM EXTENSION

The abbreviations T and F (without the periods) may be used for .TRUE. and .FALSE., respectively, (in a source program only) for the initialization of logical variables or logical arrays in the DATA statement and in the explicit type statement. For use as input/output data, see "L Format Code" under "FORMAT Statement."

END OF IBM EXTENSION

The logical constant .TRUE. or .FALSE., when assigned to a logical variable, specifies that the value of the logical variable is true or false, respectively. (See "Logical Expressions" on page 35.)

CHARACTER CONSTANTS

Definition

Character Constant—A string of any characters capable of representation in the processor. The string must be enclosed in apostrophes.

The delimiting apostrophes are not part of the data represented by the constant. An apostrophe within the character data is represented by two consecutive apostrophes with no intervening blanks. In a character constant, blanks embedded between the delimiting apostrophes are significant. The length of a character constant must be greater than zero.

Each character requires one byte of storage.

Character constants can be used in character expressions, in an assignment statement, in the argument list of a CALL statement or function reference, as data initialization values, in input or output statements, in FORMAT statements, in PARAMETER statements, or in PAUSE and STOP statements.

Valid Character Constants:	Length:
'DATA'	4
'X-COORDINATE Y-COORDINATE Z-COORDINATE'	44
'3.14'	4
'DON''T'	5

HOLLERITH CONSTANTS

Definition

Hollerith Constant—A string of any characters capable of representation in the processor and preceded by wH, where w is the number of characters in the string.

Each character requires one byte of storage.

Hollerith constants can be used only in FORMAT statements.

Valid Hollerith Constants:

```
24H INPUT/OUTPUT AREA NO. 2
6H DON'T
```

IBM EXTENSION

HEXADECIMAL CONSTANTS

Definition

Hexadecimal Constant—The character Z followed by two or more hexadecimal numbers formed from the set of characters 0 through 9 and A through F.

Hexadecimal constants may be used as data initialization values for any type of variable or array except those of character type.

One byte contains 2 hexadecimal digits. If a constant is specified as an odd number of digits, a leading hexadecimal zero is added on the left to fill the byte. The internal binary form of each hexadecimal digit is as follows:

0—0000	4—0100	8—1000	C—1100
1—0001	5—0101	9—1001	D—1101
2—0010	6—0110	A—1010	E—1110
3—0011	7—0111	B—1011	F—1111

Valid Hexadecimal Constants:

Z1C49A2F1 represents the bit string:

```
00011100010010011010001011110001
```

ZBADFADE represents the bit string:

```
00001011101011011111101011011110
```

where the first 4 zero bits are implied because an odd number of hexadecimal digits is written.

The maximum number of digits allowed in a hexadecimal constant depends upon the length specification of the variable being initialized (see "Variable Types and Lengths"). The following list shows the maximum number of digits for each length specification:

Length of Variable	Maximum Number of Hexadecimal Digits
16	32
8	16
4	8
2	4
1	2

If the number of digits is greater than the maximum, the excess leftmost hexadecimal digits are truncated; if the number of digits is less than the maximum, hexadecimal zeros are supplied on the left.

END OF IBM EXTENSION

VARIABLES

A VS FORTRAN variable is a data item, identified by a name, that occupies a storage area, except possibly in situations involving error or interruption handling where normal program flow is asynchronously interrupted. The value represented by the name is always the current value stored in the area.

Before a variable has been assigned a value, its content are undefined, and the variable should not be referred to except to assign it a value. If a variable has not been assigned a value, it does not have a predictable value.

VARIABLE NAMES

VS FORTRAN variable names must follow the rules governing element names. (See "Names" on page 8.) The use of meaningful variable names can serve as an aid in documenting a program.

Valid Variable Names:

```
B292S
RATE
```

IBM EXTENSION

```
$VAR
```

END OF IBM EXTENSION

Invalid Variable Names:

B292704	Contains more than six characters.
4ARRAY	First character is not alphabetic.
SI.X	Contains a special character.

VARIABLE TYPES AND LENGTHS

The type of a variable corresponds to the type of data the variable represents. (See Figure 5 on page 19.) Thus, an integer variable must represent integer data, a real variable must represent real data, and so on. There is no variable type associated with hexadecimal data; this type of data is identified by a name of one of the other types. There is no variable type associated with statement numbers; integer variables that contain

the statement number of an executable statement or a FORMAT statement are not considered to contain an integer variable. (See "ASSIGN Statement" on page 46.)

For every type of variable data, there is a corresponding length specification that determines the number of bytes that are reserved.

IBM EXTENSION

Optional length specification is an IBM extension.

END OF IBM EXTENSION

Figure 5 shows each data type with its associated storage length and standard length.

Data Type	Storage Length	Standard Length (Default)
Integer	[2,] 4	4
Real	4 [, 8, 16]	4
Double Precision	8	8
Complex	8 [, 16, 32]	8
Logical	[1,] 4	4
Character	1 - 500	1

Figure 5. Data Type and Storage Length

A programmer may declare the type of variable by using the following:

- Explicit specification statements
- IMPLICIT statement
- Predefined specification contained in the VS FORTRAN language

An explicit specification statement overrides an IMPLICIT statement, which, in turn, overrides the predefined specification. The optional length specification of a variable may be declared only by the IMPLICIT or explicit specification statements. If, in these statements, no length specification is stated, the default length is assumed. INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and CHARACTER are used to specify the length and type in these statements.

IBM EXTENSION

VS FORTRAN accepts INTEGER*2 to indicate 2 bytes and INTEGER*4 as an alternative to INTEGER to indicate 4 bytes; REAL*4 as an alternative to REAL to indicate 4 bytes; REAL*8 as an

alternative to DOUBLE PRECISION to indicate 8 bytes; REAL*16 to indicate 16 bytes; LOGICAL*1 to indicate 1 byte, and LOGICAL*4 as an alternative to LOGICAL to indicate 4 bytes.

END OF IBM EXTENSION

Type Declaration by the Predefined Specification

The predefined specification is a convention used to specify variables as integer or real as follows:

- If the first character of the variable name is I, J, K, L, M, or N, the variable is integer of length 4.
- If the first character of the variable name is any other alphabetic character, the variable is real of length 4.

IBM EXTENSION

- If the first character of the variable name is a currency symbol ($), the variable is real of length 4.

END OF IBM EXTENSION

This convention is the traditional FORTRAN method of specifying the type of a variable as either integer or real. Unless otherwise noted, it is presumed in the examples in this publication that this specification applies. Variables defined with this convention are of standard (default) length.

Type Declaration by the IMPLICIT Statement

The IMPLICIT statement allows a programmer to specify the type of variables in much the same way as was specified by the predefined convention. That is, the type is determined by the first character of the variable name. However, by using the IMPLICIT statement, the programmer has the option of specifying which initial characters designate a particular variable type. The IMPLICIT statement can be used to specify all types of variables—integer, real, complex, logical, and character—and to indicate storage length.

The IMPLICIT statement overrides the variable type as determined by the predefined convention.

The IMPLICIT statement is presented in greater detail in "IMPLICIT Type Statement" on page 122.

Type Declaration by Explicit Specification Statements

Explicit specification statements differ from the first two ways of specifying the type of a variable in that an explicit specification statement declares the type of a particular variable by its name rather than a group of variable names beginning with a particular letter, as specified in Figure 4 on page 8. Explicit type statements override IMPLICIT statements and the predefined specifications.

The explicit specification statements are discussed in greater detail in "Explicit Type Statement" on page 85.

ARRAY ELEMENTS

An array is an ordered and structured sequence of data items, stored as multidimensional vectors of from one to seven dimensions. The data items that make up the array are called array elements. A particular element in the array is identified by the array name and its position in the array (for example, first element, third element, seventh element, and so on). (See "Names"

on page 8.) All elements of an array have the same type and length.

To refer to any element in an array, the array name plus a parenthesized subscript must be used. In particular, the array name alone does not represent the first element except in an EQUIVALENCE statement.

Before an array element has been assigned a value, its contents is undefined, and the array element should not be referred to before assigning it a value.

SUBSCRIPTS

A subscript is a quantity (or a set of subscript expressions separated by commas) that is associated with an array name to identify a particular element of the array. The number of subscript quantities in any subscript must be the same as the number of dimensions of the array with whose name the subscript is associated. A subscript is enclosed in parentheses and is written immediately after the array name. A maximum of seven subscript expressions can appear in a subscript.

The following rules apply to the construction of subscripts. (See "VS FORTRAN Expressions" on page 25 for additional information and restrictions.)

1. Subscript expressions may contain arithmetic expressions that use any of the arithmetic operators: +, -, *, /, **.
2. Subscript expressions may contain function references that do not change any other value in the same statement.
3. Subscript expressions may contain array elements.

IBM EXTENSION

4. Mixed-mode expressions (integer and real only) within a subscript are evaluated according to normal FORTRAN rules. If the evaluated expression is real, it is converted to integer by truncation.

END OF IBM EXTENSION

5. The evaluated result of a subscript expression must always be greater than or equal to the corresponding lower dimension bound and must not exceed the corresponding upper dimension bound (see "Size and Type Declaration of an Array" on page 22 for information about dimension bounds).

Valid Array Elements:

```
ARRAY (IHOLD)

NEXT (19)

MATRIX (I-5)
```

IBM EXTENSION

```
BAK (I,J(K+2*L,.3*A(M,N)))       J is an array.
```

END OF IBM EXTENSION

```
ARRAY (I,J/4*K**2)

ARRAY (-5)

LOT (0)
```

Invalid Array Elements:

ALL(.TRUE.)	A subscript expression may not be a logical expression.
NXT (1+(1.3,2.0))	A subscript expression may not be a complex expression.

Note: The elements of an array are stored in column-major order. To step through the elements of the array in the linearized order defined as "column-major order," each subscript varies (in steps of 1) from its lowest valid value to its highest valid value, such that each subscript expression completes a full cycle before the next subscript expression to the right is incremented. Thus, the leftmost subscript expression varies most rapidly, and the rightmost subexpression varies least rapidly.

The following list is the order of an array named C defined with three dimensions:

```
DIMENSION C(1:3,1:2,1:4)

  C(1,1,1) C(2,1,1) C(3,1,1) C(1,2,1) C(2,2,1) C(3,2,1)
  C(1,1,2) C(2,1,2) C(3,1,2) C(1,2,2) C(2,2,2) C(3,2,2)
  C(1,1,3) C(2,1,3) C(3,1,3) C(1,2,3) C(2,2,3) C(3,2,3)
  C(1,1,4) C(2,1,4) C(3,1,4) C(1,2,4) C(2,2,4) C(3,2,4)
```

SIZE AND TYPE DECLARATION OF AN ARRAY

The size (number of elements) of an array is declared by specifying, in a subscript, the number of dimensions in the array and the size of each dimension. Each dimension is represented by an optional lower bound (e1) and a required upper bound (e2) in the form:

Syntax

```
name ( [e1:] e2 )
```

name
is an array name.

where:

e1
is the lower dimension bound. It is optional. If e1 (with its following colon) is not specified, its value is assumed to be 1.

e2
is the upper dimension bound and must always be specified.

The colon represents the range of values for an array's subscript. For example,

```
DIMENSION A(0:9),B(3,-2:5)

DIMENSION ARAY(-3:-1),DARY(-3:ID3**ID1)

DIMENSION IARY(3)
```

The upper and lower bounds (e1 and e2) are arithmetic expressions in which all constants and variables are of type integer.

- If the array name is an **actual** argument, the expressions can contain only constants or names of constants of type integer.
- The value of the lower bound may be positive, negative, or zero. It is assumed to be 1 if it is not specified.

- A maximum of seven dimensions is permitted. The size of each dimension is equal to the difference between the upper and lower bounds +1. If the value of the lower dimension bound is 1, the size of the dimension is equal to the value of its upper bound.
- Function or array element references are not allowed in dimension bound expressions.
- The value of the upper bound must be greater than or equal to the value of the lower bound. An upper dimension bound of an asterisk is always greater than or equal to the lower dimension bound.
- If the array name is a dummy argument and is in a subprogram, the expressions can also contain:
  - Integer variables that are also dummy arguments
  - Expressions that contain:
    - Signed or unsigned integer constants
    - Names of integer constants
    - Variables that are dummy arguments or appear in a common-block in that subprogram
- The upper dimension bound of the last dimension of a dummy array name can be an asterisk.

Size information must be given for all arrays in a VS FORTRAN program so that an appropriate amount of storage may be reserved. Declaration of this information is made by a DIMENSION statement, a COMMON statement, or by one of the explicit type specification statements. These statements are discussed in detail in alphabetic sequence in "VS FORTRAN Statement Descriptions."

The type of an array name is determined by the conventions for specifying the type of a variable name. Each element of an array is of the type and length specified for the array name.

Object-Time Dimensions

If a dummy argument array is used in a function or subroutine subprogram, the absolute dimensions of the array do not have to be explicitly declared in the subprogram by constants. Instead, the array declarators appearing in an explicit specification statement or DIMENSION statement in the subprogram may contain dummy arguments or variables in common that are integer variables of length 4 to specify the size of the array. When the subprogram is called, these integer variables receive their values from the actual arguments in the calling program reference or from common. Thus, the dimensions of a dummy array appearing in a subprogram may change each time the subprogram is called. This is called an "adjustable array" or an "object-time dimension array."

The absolute dimensions of an array must be declared in the calling program or in a higher level calling program, and the array name must be passed to the subprogram in the argument list of the calling program. The dimensions passed to the subprogram must be less than or equal to the absolute dimensions of the array declared in the calling program. The variable dimension size can be passed through more than one level of subprogram (that is, to a subprogram that calls another subprogram, passing it dimension information).

Integer variables in the explicit specification or DIMENSION statement that provide dimension information may be redefined within the subprogram but the redefinitions have no effect on the size of the array. The size of the array is determined at the entry point at which the array information is passed.

Character arrays are specified in the same manner as for the other data types. (See "DIMENSION Statement" on page 71 and "Explicit Type Statement" on page 85.) The length of each array element is either the standard length of 1 or may be declared larger with a type or IMPLICIT statement. Each character array element is treated as a single entity. Portions of an array element can be accessed through substring notation.

CHARACTER SUBSTRINGS

A character substring is a contiguous portion of a character variable or character array element. A character substring is identified by a substring reference. It may be assigned values and may be referred to. A substring reference is local to a program unit.

The form of a substring reference is:

Syntax

```
a(e1:e2)
```

a
is a character variable name or a subscripted character array name (see "Array Elements" on page 20).

e1 and **e2**
are substring expressions.

Substring expressions are optional, but the colon (:) is always required inside the parentheses. The colon represents a range of values. If e1 is omitted, a value of one is implied for e1. If e2 is omitted, a value equal to the length of the character variable or array element is implied for e2. Both e1 and e2 may be omitted; for example, the form v(:) is equivalent to v.

The value of e1 specifies the leftmost character position and the value of e2 specifies the rightmost character position of the substring. The substring information (if any) must be specified after the subscript information (if any).

- The values of e1 and e2 must be integer, positive, and nonzero.
- The value of e1 must be less than or equal to the value of e2.
- The values of e1 and e2 must be less than or equal to the number of characters contained in the corresponding variable name or array element.

Examples:

Example 1:

Given the following statements:

```
CHARACTER*5 CH(10)
CH(2)='ABCDE'
```

then

```
CH(2)(1:2) has the value AB.
CH(2)(:3)  has the value ABC.
CH(2)(3:)  has the value CDE.
```

Example 2:

```
SUBSTG(:) = SYMNAM
SUBST3(3:15) = SYMB3
SUBST5(5:9) = SUBARI(2)(1:)
```

VS FORTRAN EXPRESSIONS

VS FORTRAN provides four kinds of expressions: arithmetic, character, relational, and logical.

- The value of an arithmetic expression is always a number whose type is integer, real, or complex.
- The value of a character expression is a character string.
- The value of a relational or logical expression is always a logical value: .TRUE. or .FALSE..

EVALUATION OF EXPRESSIONS

VS FORTRAN expressions are evaluated according to the following rules:

- Any variable, array element, function, or character substring referred to as an operand in an expression must be defined (that is, must have been assigned a value) at the time the reference is executed.

  In an expression, an integer operand must be defined with an integer value, rather than a statement number. (See "ASSIGN Statement" on page 46.) If a character string or a substring is referred to, all of the characters referred to must be defined at the time the reference is executed.

- The execution of a function reference in a statement must not alter the value of any other entity within the statement in which the function reference appears. The execution of a function reference in a statement must not alter the value of any entity in COMMON that affects the value of any other function reference in that statement.

  If a function reference in a statement alters the value of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the statement. For example, the following statements are prohibited if the reference to the function F defines I or if the reference to the function G defines X:

  ```
  A(I) = F(I)

  Y = G(X) + X
  ```

  The data type of an expression in which a function reference appears does not affect the evaluation of the actual arguments of the function.

- Any array element reference requires the evaluation of its subscript. The data type of an expression in which an array reference appears does not affect, nor is it affected by, the evaluation of the subscript.
- Any execution of a substring reference requires the evaluation of its substring expressions. The data type of an expression in which a substring name appears does not affect, nor is it affected by, the evaluation of the substring expressions.

ARITHMETIC EXPRESSIONS

The simplest arithmetic expression consists of a primary, which may be a single constant, name of a constant, variable, array

element, function reference, or another expression enclosed in parentheses. The primary may be either integer, real, or complex.

In an expression consisting of a single primary, the type of the primary is the type of the expression. Examples of arithmetic expressions are shown in Figure 6.

Primary	Type of Primary	Type	Length
3	Integer constant	Integer	4
A	Real variable	Real	4
3.14D3	Real constant	Real	8
3.14D3	Double precision constant	Double precision	8
(2.0,5.7)	Complex constant	Complex	8
SIN(X)	Real function reference	Real	4
(A*B+C)	Parenthesized real expression	Real	4

Figure 6. Examples of Arithmetic Expressions

ARITHMETIC OPERATORS

More complicated arithmetic expressions containing two or more primaries may be formed by using arithmetic operators that express the computation(s) to be performed.

The arithmetic operators are shown in Figure 7.

Arithmetic Operator	Definition
**	Exponentiation
*	Multiplication
/	Division
+	Addition (or unary plus)
-	Subtraction (or unary minus)

Figure 7. Arithmetic Operators

RULES FOR CONSTRUCTING ARITHMETIC EXPRESSIONS

The following are the rules for constructing arithmetic expressions that contain arithmetic operators:

- All desired computations must be specified explicitly. That is, if more than one primary appears in an arithmetic expression, they must be separated from one another by an arithmetic operator. For example, the two variables A and B are not multiplied if written:

  AB

  In fact, AB is regarded as a single variable with a two-letter name.

If multiplication is desired, the expression must be written as follows:

```
A*B or B*A
```

- No two arithmetic operators may appear consecutively in the same expression. For example, the following expressions are invalid:

  ```
  A*/B and A*-B
  ```

  The expression A*-B could be written correctly as

  ```
  A*(-B)
  ```

  Two asterisks (**) designate exponentiation, not two multiplication operations.

- Order of Computation

  In the evaluation of expressions, priority of the operations is shown in Figure 8.

Operation	Hierarchy
Evaluation of functions	1st
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th

Figure 8. Hierarchy of Arithmetic Operations

Note: A unary plus or minus has the same hierarchy as a plus or minus in addition or subtraction.

If two or more operators of the same priority appear successively in the expression, the order of priority of those operators is from left to right, except for successive exponentiation operators, where the evaluation is from right to left.

Consider the evaluation of the expression in the assignment statement:

```
RESULT= A*B+C*D**I
```

1. A*B Call the result X (multiplication) (X+C*D**I)
2. D**I Call the result Y (exponentiation) (X+C*Y)
3. C*Y Call the result Z (multiplication) (X+Z)
4. X+Z Final operation (addition)

The expression:

```
A**B**C
```

is evaluated as follows:

1. B**C Call the result Z.
2. A**Z Final operation.

Expressions with a unary minus are treated as follows:

A=-B is treated as A=0-B	
A=-B*C is treated as A=-(B*C)	Because * has higher precedence than -
A=-B+C is treated as A=(-B)+C	Because - has equal precedence to +

USE OF PARENTHESES IN ARITHMETIC EXPRESSIONS

Because the order of evaluation (and, consequently, the result) of an expression can be changed through the use of parentheses, refer to Figure 9, Figure 10, and Figure 11 to determine the type and length of intermediate results. Where parentheses are used, the expression contained within the most deeply nested parentheses (that is, the innermost pair of parentheses) is evaluated first. A parenthesized expression is considered a primary.

For example, the expression,

```
B/((A-B)*C)+A**2
```

is effectively evaluated in the following order:

1.	A-B	Call the result W	B/(W*C)+A**2
2.	W*C	Call the result X	B/X+A**2
3.	B/X	Call the result Y	Y+A**2
4.	A**2	Call the result Z	Y+Z
5.	Y+Z	Final operation	

TYPE AND LENGTH OF THE RESULT OF ARITHMETIC EXPRESSIONS

The type and length of the result of an operation depend upon the type and length of the two operands (primaries) involved in the operation.

Figure 9 shows the type and length of the result of adding, subtracting, multiplying, or dividing when the first operand is an integer.

Figure 10 shows the type and length of the result of adding, subtracting, multiplying, or dividing when the first operand is real.

Figure 11 shows the type and length of the result of adding, subtracting, multiplying, or dividing when the first operand is complex.

Note: Except for a value raised to an integer power, if two operands are of different type and length, the operand that differs from the type and/or length of the result is converted to the type and/or length of the result. Thus the operator operates on a pair of operands of matching type and length.

When an operand of real or complex type is raised to an integer power, the integer operand is not converted. The resulting type and length match the type and length of the base.

First Operand / Second Operand	Integer (2)	Integer (4)
Integer (2)	Integer (2)	Integer (4)
Integer (4)	Integer (4)	Integer (4)
Real (4)	Real (4)	Real (4)
Real (8)	Real (8)	Real (8)
Real (16)	Real (16)	Real (16)
Complex (8)	Complex (8)	Complex (8)
Complex (16)	Complex (16)	Complex (16)
Complex (32)	Complex (32)	Complex (32)

Figure 9. Type and Length where the First Operand is Integer

First Operand / Second Operand	Real (4)	Real (8)	Real (16)
Integer (2)	Real (4)	Real (8)	Real (16)
Integer (4)	Real (4)	Real (8)	Real (16)
Real (4)	Real (4)	Real (8)	Real (16)
Real (8)	Real (8)	Real (8)	Real (16)
Real (16)	Real (16)	Real (16)	Real (16)
Complex (8)	Complex (8)	Complex (16)	Complex (32)
Complex (16)	Complex (16)	Complex (16)	Complex (32)
Complex (32)	Complex (32)	Complex (32)	Complex (32)

Figure 10. Type and Length where the First Operand is Real

First Operand / Second Operand	Complex (8)	Complex (16)	Complex (32)
Integer (2)	Complex (8)	Complex (16)	Complex (32)
Integer (4)	Complex (8)	Complex (16)	Complex (32)
Real (4)	Complex (8)	Complex (16)	Complex (32)
Real (8)	Complex (16)	Complex (16)	Complex (32)
Real (16)	Complex (32)	Complex (32)	Complex (32)
Complex (8)	Complex (8)	Complex (16)	Complex (32)
Complex (16)	Complex (16)	Complex (16)	Complex (32)
Complex (32)	Complex (32)	Complex (32)	Complex (32)

Figure 11. Type and Length where the First Operand is Complex

EXAMPLES OF ARITHMETIC EXPRESSIONS

Assume that the type of the following variables has been specified as indicated below:

Name	Type	Length
I, J, K	Integer variables	4 [, 2, 2]
C	Real variable	4
D	Complex variable	[16]

Then the expression I*J/C**K+D is evaluated as follows:

Subexpression		Type and Length
I*J	(Call the result X)	Integer of length 4
C**K	(Call the result Y)	Real of length 4
X/Y	(Call the result Z)	Real of length 4

(X is converted to real of length 4 before division is performed.)

IBM EXTENSION

Z+D	Complex of length 16

(Z is expanded to real of length 8 and a complex quantity of length 16 (call it W) is formed in which the real part is the expansion of Z and the imaginary part is zero. Then the real part of W is added to the real part of D and the imaginary part of W is added to the imaginary part of D.)

Thus, the final type of the entire expression is complex of length 16, but the types of the intermediate expressions change at different stages in the evaluation.

END OF IBM EXTENSION

Depending on the values of the variables involved, the result of the expression I*J*C might be different from I*C*J. This may occur because of the number of conversions performed during the evaluation of the expression.

Because the operators are the same, the order of the evaluation is from left to right. With I*J*C, a multiplication of the two integers I*J yields an intermediate result of type integer and length 4. This intermediate result is converted to a type real of length 4 and multiplied with C of type real of length 4 to yield a type real of length 4 result.

With I*C*J, the integer I is converted to a type real of length 4 and the result is multiplied with C of type real of length 4 to yield an intermediate result of type real of length 4. The integer J is converted to a type real of length 4 and the result is multiplied with the intermediate result to yield a type real of length 4 result.

Evaluation of I*J*C requires one conversion and I*C*J requires two conversions. The expressions require that the computation be performed with different types of arithmetic. This may yield different results.

When division is performed using two integers, any remainder is truncated (without rounding) and an integer quotient is given. If the mathematical quotient is less than 1, the answer is 0. The sign is determined according to the rules of algebra. For example:

I	J	I/J
9	2	4
-5	2	-2
1	-4	0

CHARACTER EXPRESSIONS

The simplest form of a character expression is a character constant, character array element reference, character substring reference, or character function reference. More complicated character expressions may be formed by using one or more character operands together with character operators and parentheses.

The character operator is shown in Figure 12.

Character Operator	Definition
//	Concatenation

Figure 12. Character Operator

The concatenation operation joins the operands in such a way that the last character of the operand to the left immediately precedes the first character of the operand to the right. For example:

'AB'//'CD' yields the value of 'ABCD'

The result of a concatenation operation is a character string consisting of the values of the operands concatenated left to right and its length is equal to the sum of the lengths of the operands.

Note: Except in a CHARACTER assignment statement, the operands of a concatenation operation must not have inherited length. That is, their length specification must not be an asterisk (*) unless the operand is the name of a constant. See "Explicit Type Statement" on page 85.

USE OF PARENTHESES IN CHARACTER EXPRESSIONS

Parentheses have no effect on the value of a character expression. For example:

If X has the value 'AB',

Y has the value 'CD'

and

Z has the value 'EF'

then the two expressions:

X//Y//Z

X//(Y//Z)

both yield the same result, the value 'ABCDEF'

Valid Character Expressions:

Substring:

```
ST1311(I) = CVAR1(:I)
```

Function Reference:

```
ST1314(IVAR1) = CHAR(IVAR1)
```

RELATIONAL EXPRESSIONS

Relational expressions are formed by combining two arithmetic expressions with a relational operator or two character expressions with a relational operator.

The six relational operators are shown in Figure 13.

Relational Operator	Definition
.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to

Figure 13. Relational Operators

Relational operators:

- Express a condition that can be either true or false.
- Operators may be used to compare two arithmetic expressions (except complex) or two character expressions. Only the .EQ. and .NE. operators may be used to compare an arithmetic expression with a complex expression. If the two arithmetic expressions being compared are not of the same type or length, they are converted following the rules indicated in Figure 9, Figure 10, and Figure 11.
- Comparison of an arithmetic expression to a character expression or vice versa is not allowed.

In the case of character expressions, the shorter operand is considered as being extended temporarily on the right with blanks to the length of the longer operand. The comparison is made from left to right, character by character, according to the collating sequence as shown in Figure 4 and in "Appendix E. EBCDIC and ASCII Codes."

Examples:

Assume that the type of the following variables has been specified as indicated:

Variable Names	Type
ROOT, E	Real
A, I, F	Integer
L	Logical
C	Complex
CHAR	Character of length 10

Then the following examples illustrate valid and invalid relational expressions.

Valid Relational Expressions:

```
E .LT. I

E**2.7 .LE. (5*ROOT+4)

.5 .GE. (.9*ROOT)

E .EQ. 27.3E+05

CHAR .EQ. 'ABCDEFGH'

C.NE. CMPLX(ROOT,E)
```

Invalid Relational Expressions:

C.GE.(2.7,5.9E3)	Complex quantities can only be compared for equal or not equal in relational expressions.
L.EQ.(A+F)	Logical quantities may never be compared by relational operators.
E**2 .LT 97.1E1	There is a missing period immediately after the relational operator.
.GT.9	There is a missing arithmetic expression before the relational operator.
E*2 .EQ. 'ABC'	A character expression may not be compared to an arithmetic expression.

IBM EXTENSION

Length of a Relational Expression: A relational expression is always evaluated to a LOGICAL*4 result, but the result can be converted in an assignment statement to LOGICAL*1.

END OF IBM EXTENSION

LOGICAL EXPRESSIONS

The simplest form of logical expression consists of a single logical primary. A logical primary can be a logical constant, a name of a logical constant, a logical variable, a logical array element, a logical function reference, a relational expression (which may be an arithmetic relational expression or a character relational expression), or a logical expression enclosed in parentheses. A logical primary, when evaluated, always has a value of true or false.

More complicated logical expressions may be formed by using logical operators to combine logical primaries.

LOGICAL OPERATORS

The logical operators are shown in Figure 14. (A and B represent logical constants or variables, or expressions containing relational operators.)

Logical Operator	Use	Meaning
.NOT.	.NOT.A	If A is true, then .NOT.A is false; if A is false, then .NOT.A is true.
.AND.	A.AND.B	If A and B are both true, then A.AND.B is true; if either A or B or both are false, then A.AND.B is false.
.OR.	A.OR.B	If either A or B or both are true, then A.OR.B is true; if both A and B are false, then A.OR.B is false.
.EQV.	A.EQV.B	If A and B are both true or both false, then A.EQV.B is true; otherwise it is false.
.NEQV.	A.NEQV.B	If A and B are both true or both false, then A.NEQV.B is false; otherwise it is true.

Figure 14. Logical Operators

The only valid sequences of two logical operators are:

```
.AND..NOT.
.OR..NOT.
.EQV..NOT.
.NEQV..NOT.
```

The sequence .NOT..NOT. is invalid.

Only those expressions which have a value of true or false when evaluated, may be combined with the logical operators to form logical expressions.

Examples:

Assume that the types of the following variables have been specified as indicated:

Variable Names	Type
ROOT, E	Real
A, I, F	Integer
L, W	Logical
CHAR, SYMBOL	Character of lengths 3 and 6, respectively

Then the following examples illustrate valid and invalid logical expressions using both logical and relational operators.

Valid Logical Expressions:

```
(ROOT*A .GT. A) .AND. W
L .AND. .NOT. (I .GT. F)
(E+5.9E2 .GT. 2*E) .OR. L
.NOT. W .AND. .NOT. L
L .AND. .NOT. W .OR. CHAR//'123'.LT.SYMBOL
(A**F .GT. ROOT .AND. .NOT. I .EQ. E)
```

Invalid Logical Expressions:

A.AND.L	A is not a logical expression.
.OR.W	.OR. must be preceded by a logical expression.
NOT.(A.GT.F)	Missing period before the logical operator .NOT..
L.AND..OR.W	The logical operators .AND. and .OR. must always be separated by a logical expression.
.AND.L	.AND. must be preceded by a logical expression.

ORDER OF COMPUTATIONS IN LOGICAL EXPRESSIONS

In the evaluation of logical expressions, priority of operations involving arithmetic operators is as shown in Figure 15. Within a hierarchic level, computation is performed from left to right.

Operation Involving Arithmetic Operators	Hierarchy
Evaluation of functions	1st (highest)
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th
Relationals (.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.)	5th
.NOT.	6th
.AND.	7th
.OR.	8th
.EQV. or .NEQV.	9th

Figure 15. Hierarchy of Operations Involving Arithmetic Operators

In the evaluation of logical expressions, priority of operations involving character operators is as shown in Figure 16. Within a hierarchic level, computation is performed from left to right.

Operation Involving Character Operators	Hierarchy
Evaluation of functions	1st (highest)
Concatenation (//)	2nd
Relationals (.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.)	3th
.NOT.	4th
.AND.	5th
.OR.	6th
.EQV. or .NEQV.	7th

Figure 16. Hierarchy of Operations Involving Character Operators

Example:

Assume the type of the following variables has been specified as follows:

Variable Names	Type	Length
B,D	REAL	4
A	REAL	8
L,N	LOGICAL	4

The expression

```
A.GT.D**B.AND..NOT.L.OR.N
```

is effectively evaluated in the following order (and from left to right):

1. D**B Call the result W.

   Exponentiation is performed because arithmetic operators have a higher priority than relational operators, yielding a real result W of length 4.

2. A.GT.W Call the result X.

   The real variable A of length 8 is compared to the real variable W which was extended to a length of 8, yielding a logical result X whose value is true or false.

3. .NOT.L Call the result Y.

   The logical negation is performed because .NOT. has a higher priority than .AND., yielding a logical result Y whose value is true if L is false and false if L is true.

4. X.AND.Y Call the result Z.

   The logical operator .AND. is applied because .AND. has a higher priority than .OR. to yield a logical result Z whose value is true if both X and Y are true and false, if both X and Y are false, or if either X or Y is false.

5. Z.OR.N

   The logical operator .OR. is applied to yield a logical result of true if either Z or N is true or if both Z and N are true. If both Z and N are false, the logical result is false.

Note: Calculating the value of logical expressions may not always require that all parts be evaluated. Functions within logical expressions may or may not be invoked. For example, assume a logical function called LGF. In the expression A.OR.LGF(.TRUE.), it should not be assumed that the LGF function is always invoked, since it is not always necessary to do so to evaluate the expression when A is true.

USE OF PARENTHESES IN LOGICAL EXPRESSIONS

Parentheses may be used in logical expressions to specify the order in which the operations are to be performed. Where parentheses are used, the expression contained within the most deeply nested parentheses (that is, the innermost pair of parentheses) is evaluated first.

Example:

Assume the type of the following variables specified as follows:

Variable Names	Type	Length
B	REAL	4
C	REAL	8
K,L	LOGICAL	4

The expression

```
.NOT.((B.GT.C.OR.K).AND.L)
```

is evaluated in the following order:

1. B.GT.C Call the result X.

   B is extended to a real of length 8 and compared with C of length 8 yielding a logical result X of length 4 whose value is true if B is greater than C or false if B is less than or equal to C.

2. X.OR.K Call the result Y.

   The logical operator .OR. is applied to yield a logical result of Y whose value is true if either X or K is true or if both X and K are true. If both X and K are false, the logical result Y is false.

3. Y.AND.L Call the result Z.

   The logical operator .AND. is applied to yield a logical result Z whose value is true if both Y and L are true and false if both Y and L are false or if either Y or L is false.

4. .NOT.Z

   The logical negation is performed to yield a logical result whose value is true if Z is false and false if Z is true.

A logical expression to which the logical operator .NOT. applies must be enclosed in parentheses if it contains two or more quantities. Otherwise, because of the higher precedence of the .NOT. operator, it will apply to the first operand of the relation. For example, assume that the values of the logical variables, A and B, are false and true, respectively. Then the following two expressions are not equivalent:

```
.NOT.(A.OR.B)

.NOT.A.OR.B
```

In the first expression, A.OR.B is evaluated first. The result is true; but .NOT.(.TRUE.) is the equivalent of .FALSE.. Therefore, the value of the first expression is false.

In the second expression, .NOT.A is evaluated first. The result is true; but .TRUE..OR.B is the equivalent of .TRUE.. Therefore, the value of the second expression is true.

The lengths of the results of the various logical operations are shown in Figure 17. (The result of logical operations is always logical of length 4.)

Second Operand \ First Operand	Logical (1)	Logical (4)
Logical (1)	Logical (4)	Logical (4)
Logical (4)	Logical (4)	Logical (4)

Figure 17. Type and Length of the Result of Logical Operations

VS FORTRAN STATEMENTS

Source programs consist of a set of statements from which the compiler generates machine instructions and allocates storage for data areas. A given VS FORTRAN statement performs one of three functions:

- It performs certain executable operations (for example, addition, multiplication, branching).
- It specifies the nature of the data being handled.
- It specifies the characteristics of the source program.

VS FORTRAN statements are either executable or nonexecutable.

VS FORTRAN STATEMENT CATEGORIES

Statements are divided into the following categories according to what they do:

- Assignment statements
- Control statements
- Data statement

IBM EXTENSION

- Debug statements

END OF IBM EXTENSION

- Input/output statements
- Main program statement
- Specification statements
- Subprogram statements

IBM EXTENSION

- VS FORTRAN compiler directives

END OF IBM EXTENSION

ASSIGNMENT STATEMENTS

There are four types of assignment statements: the arithmetic, character, and logical assignment statements and the ASSIGN statement. Execution of an assignment statement assigns a value to a variable. Assignment statements are executable.

CONTROL STATEMENTS

In the absence of control statements, VS FORTRAN statements are executed sequentially. That is, after one statement has been executed, the statement immediately following it is executed. Control statements alter this normal sequence of execution of statements in the program. They are executable.

CALL	IF (ELSE, ELSE IF, END IF)
CONTINUE	PAUSE
DO	RETURN
END	STOP
GO TO	

DATA STATEMENT

The DATA statement assigns initial values to variables, array elements, arrays, and substrings. It is nonexecutable.

IBM EXTENSION

DEBUG STATEMENTS

The debug facility is a programming aid that helps locate errors in a VS FORTRAN source program. The debug facility traces the flow of execution within a program, traces the flow of execution between programs, displays the values of variables and arrays, and checks the validity of subscripts. DISPLAY, TRACE OFF, and TRACE ON are executable; AT, DEBUG, and END DEBUG are nonexecutable.

AT	END DEBUG
DEBUG	TRACE OFF
DISPLAY	TRACE ON

END OF IBM EXTENSION

INPUT/OUTPUT STATEMENTS

Input/output (I/O) statements transfer data between two areas of internal storage or between internal storage and an input/output device. Examples of input/output devices are card readers, printers, punches, magnetic tapes, disk storage units, or terminals.

The I/O statements allow the programmer to specify how to process the VS FORTRAN files at different times during the execution of a program. Except for the FORMAT statements, these statements are executable.

BACKSPACE	OPEN
CLOSE	PRINT
ENDFILE	READ
FORMAT	REWIND
INQUIRE	WRITE

IBM EXTENSION

WAIT

END OF IBM EXTENSION

Note: The description of the VS FORTRAN input and output statements is made from the point of view of a VS FORTRAN program. Therefore words such as file, record, or OPEN must not be confused with the same words used when discussing an operating system. See the descriptions of each I/O statement.

MAIN PROGRAM STATEMENT

The PROGRAM statement names the main program. It can only be used in a main program. It is not required. The PROGRAM statement is nonexecutable.

SPECIFICATION STATEMENTS

The specification statements provide the compiler with information about the nature of the data in the source program. In addition, they supply the information required to allocate storage for this data.

The specification statements must follow the PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statements. They may be preceded by FORMAT or ENTRY statements. Specification statements are nonexecutable.

- COMMON
- DIMENSION
- EQUIVALENCE
- Explicit type:
  - COMPLEX, INTEGER,
  - LOGICAL, REAL,
  - CHARACTER, and
  - DOUBLE PRECISION
- EXTERNAL
- IMPLICIT
- INTRINSIC
- PARAMETER
- SAVE

IBM EXTENSION

- NAMELIST

END OF IBM EXTENSION

SUBPROGRAM STATEMENTS

There are three subprogram statements: FUNCTION, SUBROUTINE, and BLOCKDATA. There are also intrinsic function procedures and statement function procedures. The list of intrinsic functions supplied with VS FORTRAN is contained in "Appendix B. FORTRAN-Supplied Procedures" on page 204.

Function subprograms differ from subroutine subprograms in the way they are invoked and in that function subprograms return a value to the calling program, whereas subroutine subprograms need not return any.

The function subprogram is a VS FORTRAN subprogram that begins with a FUNCTION statement. It is independently written and is executed whenever its name is appropriately referred to in another program. It is called by coding its name with any necessary parameters. At least one executable statement in the function subprogram must assign a result to the function name; this value is returned to the calling program as the result of the function.

The subroutine subprogram is similar to the function subprogram except that it begins with a SUBROUTINE statement and does not return an explicit result to the calling program. The rules for naming function and subroutine subprograms are similar. They both require an END statement and they both may contain dummy arguments. Like the function subprogram, the subroutine subprogram can be a set of commonly used computations, but it need not return any results to the calling program. The subroutine subprogram is executed whenever its name is referred to by the CALL statement.

Subprogram statements are nonexecutable.

- BLOCK DATA
- ENTRY
- FUNCTION
- Statement function
- SUBROUTINE

IBM EXTENSION

VS FORTRAN COMPILER DIRECTING STATEMENTS

The EJECT and INCLUDE statements are IBM extensions that direct the compiler to start a new page or to insert one or more source statements into the program. They are not considered part of the VS FORTRAN language.

END OF IBM EXTENSION

ORDER OF STATEMENTS IN A PROGRAM UNIT

The order of statements in a VS FORTRAN program unit (other than a BLOCK DATA subprogram) is as follows:

1. PROGRAM or subprogram statement, if any.
2. PARAMETER statements and/or IMPLICIT statements, if any.
3. Other specification statements, if any. (Explicit specification statements that initialize variables or arrays must follow other specification statements that contain the same variable or array names.)
4. Statement function definitions, if any.
5. Executable statements.
6. END statement.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. Any specification statement that specifies the type of a name of a constant must precede the PARAMETER statement that defines that particular name of a constant; the PARAMETER statement must precede all other statements containing the names of constants that are defined in the PARAMETER statement.

FORMAT and ENTRY statements may appear anywhere after the PROGRAM or subprogram statement and before the END statement. The ENTRY statement, however, may not appear between a block IF statement and its corresponding END IF statement or within the range of a DO. DATA statements must follow the IMPLICIT statements and any specification statements that contain the same variable or array names.

IBM EXTENSION

A NAMELIST statement declaring a NAMELIST name must precede the use of that name in any input/output statement. Its placement is as indicated for other specification statements.

END OF IBM EXTENSION

The order of statements in BLOCK DATA subprograms is discussed in "BLOCK DATA Statement" on page 56. Figure 18 shows a diagram of the order of statements.

- The vertical lines in the figure delineate varieties of statements that may be interspersed. For example, FORMAT statements may be interspersed with statement function statements and executable statements.
- Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements.

<table>
<tr><td rowspan="5">Comment Lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
Statements</td><td rowspan="2">PARAMETER Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="4">END Statement</td></tr>
</table>

Figure 18. Order of Statements and Comment Lines

VS FORTRAN STATEMENT DESCRIPTIONS

The rules for coding each VS FORTRAN statement are described in this section in alphabetic sequence. Brief examples are included. For additional examples and explanations, see VS FORTRAN Application Programming: Guide.

Notes:

1. Comments and statement numbers are included because, although they are not actual statements, they are integral parts of VS FORTRAN programs.
2. Each described statement begins at the top of a page.

ARITHMETIC IF STATEMENT

See "IF Statements" on page 117.

ASSIGN STATEMENT

The ASSIGN statement assigns a number (**statement number**) to an integer variable. See also "Statement Numbers" on page 171.

Syntax

```
ASSIGN stn TO i
```

stn
is the number of an executable statement or a FORMAT statement in the program unit containing the ASSIGN statement.

i
is the name of an integer variable (not an array element) of length 4 that is assigned the statement number stn.

The statement number must be the number of a statement that appears in the same program unit as the ASSIGN statement. The statement number must be the number of an executable statement or a FORMAT statement.

Execution of ASSIGN is the only way that a variable can be defined with a statement number.

A variable must have been defined with a statement number when it is referred to in an assigned GO TO statement or as a format identifier in an input or output statement. An integer variable defined with a statement number may be redefined with the same or a different statement number or an integer value.

If stn is the statement number of an executable statement, i can be used in an assigned GOTO statement.

If stn is the statement number of a FORMAT statement, i can be used as the format identifier in a READ, WRITE, or PRINT statement with FORMAT control.

The value of i is not the integer constant represented by stn and cannot be used as such. To use i as an integer, it must be assigned an integer value by an assignment or input statement. This assignment can be done directly or through EQUIVALENCE, COMMON, or argument passing.

ASSIGNED GO TO STATEMENT

See "GO TO Statements" on page 115.

ASSIGNMENT STATEMENTS

This VS FORTRAN statement closely resembles a conventional algebraic equation; however, the equal sign specifies a replacement operation rather than equality. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable, array element, character substring, or character variable to the left of the equal sign.

Syntax

```
a = b
```

a
is a variable, array element, character substring, or character variable.

b
is an arithmetic, logical, or character expression.

An assignment statement is used for the results of calculations. The result of evaluating the expression replaces the current value of a designated variable, array element, or character substring. There are three assignment statements: arithmetic, logical, and character.

Arithmetic Assignment Statement

If b is an arithmetic expression, a must be an integer, real, or complex variable or an array element.

Figure 19 shows the rules for conversion in arithmetic assignment statements, a=b, where the type of b is integer or real.

Figure 20 shows the rules for conversion in arithmetic assignment statements, a=b, where the type of b is complex.

Character Assignment Statement

If b is a character expression, a must be a character variable, character array element, or character substring.

None of the character positions being defined in a must be referenced in b directly or through associations of variables (that is, using COMMON, EQUIVALENCE, or argument passing).

The lengths of a and b may be different. The characters of b are moved from left to right into the corresponding character positions of a. If a has more positions than there are characters in b, the rightmost positions of a are filled with blanks. If a has fewer positions than there are characters in b, only the leftmost characters of b are moved to fill the positions of a.

Logical Assignment Statement

If b is a logical expression, a must be a logical variable or logical array element. The value of b must be either true or false.

Type of a \ Type of b	INTEGER*2 INTEGER*4 INTEGER	REAL*4 REAL	REAL*8 DOUBLE PRECISION	REAL*16
INTEGER*2 INTEGER*4 INTEGER	Assign	Fix and assign	Fix and assign	Fix and assign
REAL*4 REAL	Float and assign	Assign	Real assign	Real assign
REAL*8 DOUBLE PRECISION	DP float and assign	DP extend and assign	Assign	DP assign
REAL*16	QP float and assign	QP extend and assign	QP extend and assign	Assign
COMPLEX*8 COMPLEX	Float and assign to real part; imaginary part set to 0	Assign to real part; imaginary part set to 0	Real assign real part; imaginary part set to 0	Real assign real part; imaginary part set to 0
COMPLEX*16	DP float and assign to real part; imaginary part set to 0	DP extend and assign to real part; imaginary part set to 0	Assign to real part; imaginary part set to 0	DP assign real part; imaginary part set to 0
COMPLEX*32	QP float and assign to real part; imaginary part set to 0	QP extend and assign to real part; imaginary part set to 0	QP extend and assign real part; imaginary part set to 0	Assign real part; imaginary part set to 0

Figure 19. Conversion Rules for the Arithmetic Assignment Statement a=b Where Type of b is Integer or Real

Type of a \ Type of b	COMPLEX*8 COMPLEX	COMPLEX*16	COMPLEX*32
INTEGER*2 INTEGER*4 INTEGER	Fix and assign real part; imaginary part not used	Fix and assign real part; imaginary part not used	Fix and assign real part; imaginary part not used
REAL*4 REAL	Assign real part; imaginary part not used	Real assign, real part; imaginary part not used	Real assign, real part; imaginary part not used
REAL*8 DOUBLE PRECISION	DP extend and assign real part; imaginary part not used	Assign real part; imaginary part not used	DP assign real part; imaginary part not used
REAL*16	QP extend and assign real part; imaginary part not used	QP extend and assign real part; imaginary part not used	Assign real part; imaginary part not used
COMPLEX*8 COMPLEX	Assign	Real assign real and imaginary parts	Real assign real and imaginary parts
COMPLEX*16	DP extend and assign real and imaginary parts	Assign	DP assign real and imaginary parts
COMPLEX*32	QP extend and assign real and imaginary parts	QP extend and assign real and imaginary parts	Assign

Figure 20. Conversion Rules for the Arithmetic Assignment Statement a=b Where Type of b is Complex

Notes to Figures: IBM extensions are shown with inner boxes in the figures. For clarity of presentation, the extensions are not marked in the following definitions. Terms in the figures are defined as follows:

Assign Transmit the expression value without change. If the expression value contains more significant digits than the variable a can hold, the value assigned to a is unpredictable.

Real assign Transmit to a as much precision of the most significant part of the expression value as REAL*4 data can contain.

DP assign Transmit as much precision of the most significant part of the expression value as double precision (REAL*8) data can contain.

Fix Truncate the fractional portion of the expression value and transform the result to an integer of length 4 bytes. If the expression value contains more significant digits than an integer of length 4 bytes can hold, the value assigned to the integer variable is unpredictable.

Float Transform the integer expression value to a REAL*4 number, retaining in the process as much precision of the value as a REAL*4 number can contain.

DP float Transform the integer expression value to a double precision (REAL*8) number.

DP extend Extend the real value to a double precision (REAL*8) number.

QP float Transform the integer expression value to a REAL*16 number.

QP extend Extend the real value to a REAL*16 number.

Examples:

Assume the type of the following data items has been specified:

Name	Type	Length
I, J, K	Integer variables	4, 4 [, 2]
A, B, C, D	Real variables	4, 4, 8, 8
E	Complex variable	8
F(1),...,F(5)	Real array elements	4
G, H	Logical variables	4, 4

The following examples illustrate valid assignment statements using constants, variables, and array elements as defined above.

Statement	Description
A = B	The value of A is replaced by the current value of B.
K = B	The value of B is converted to an integer value, and the value of K is replaced by as much as can be held in 2 bytes.
A = I	The value of I is converted to a real value, and replaces the value of A.
I = I + 1	The value of I is replaced by the value of I + 1.
E = I**J+D	I is raised to the power J and the result is converted to a real value to which the value of D is added. This result replaces the real part of the complex variable E. The imaginary part of the complex variable is set to zero.
A = C*D	The most significant part of the product of C and D replaces the value of A.
A = E	The real part of the complex variable E replaces the value of A.
E = A	The value of A replaces the value of the real part of the complex variable E; the imaginary part is set equal to zero.
G = .TRUE.	The value of G is replaced by the logical value true.
H = .NOT.G	If G is true, the value of H is replaced by the logical value false. If G is false, the value of H is replaced by the logical value true.
G = 3..GT.I	The value of I is converted to a real value; if the real constant 3. is greater than this result, the logical value true replaces the value of G. If 3. is not greater than the converted I, the logical value false replaces the value of G.
E = (1.0,2.0)	The value of the complex variable E is replaced by the value of the complex constant (1.0,2.0). The statement E = (A,B), where A and B are real variables, is invalid. The mathematical function subprogram CMPLX can be used for this purpose. See "Appendix B. FORTRAN-Supplied Procedures" on page 204.
F(1) = A	The value of element 1 of array F is replaced by the value of A.
E = F(5)	The real part of the complex constant E is replaced by the value of array element F(5). The imaginary part is set equal to zero.

Statement	Description
C = 99999999.0	Even though C is of length 8, the constant having no exponent is considered to be of length 4. Thus the number will not have the accuracy that may be intended. If the basic real constant were entered as 99999999.0D0, the converted value placed in the variable C would be a closer approximation to the entered basic real decimal constant because 15 decimal digits can be represented in 8 bytes.
ST1306(1:20) = 'TEST'//CHAR1	CHAR1 must be declared CHARACTER with a type statement.

IBM EXTENSION

AT STATEMENT

The AT statement identifies the beginning of a debug packet and indicates the point in the program at which debugging is to begin.

Syntax

```
AT stn
```

stn
is the number of an executable statement in the program or function or subroutine subprogram to be debugged.

The debugging operations specified within the debug packet are performed prior to the execution of the statement indicated by the statement number (stn) in the AT statement.

The statement number cannot be specified in another debug packet.

There must be one AT statement for each debug packet; there may be many debug packets for one program or subprogram.

The AT statement identifies the beginning of a debug packet and the end of the preceding packet (if any) unless this is the last packet, in which case it is ended by the END DEBUG statement.

For a more complete discussion of debug packets and for examples of the AT statement, see "DEBUG Statement" on page 68.

END OF IBM EXTENSION

BACKSPACE STATEMENT

The BACKSPACE statement positions a sequentially accessed external file at the beginning of the VS FORTRAN record last written or read. (See "OPEN Statement" on page 134.)

Syntax

```
BACKSPACE un

BACKSPACE ( [UNIT=]un [,IOSTAT=ios] [,ERR=stn] )
```

UNIT=un
un is the reference to the number of an I/O unit. It can optionally be preceded by UNIT= if the second form of the statement is used. un can be an integer or real arithmetic expression. Its value (after conversion to integer of length 4, if necessary) must be zero or positive; otherwise, an error is detected.

If UNIT= is not specified, un must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

IOSTAT=ios
is optional. ios is an integer variable or an integer array element of length 4. ios is set positive if an error is detected; it is set to zero if no error is detected. VSAM return and reason codes are placed in ios.

ERR=stn
stn is the number of a statement in the same program unit as the BACKSPACE statement. Transfer is made to stn if an error is detected.

Valid BACKSPACE Statements:

```
BACKSPACE un

BACKSPACE (un,ERR=stn)

BACKSPACE (UNIT=un,IOSTAT=ios,ERR=stn)

BACKSPACE (ERR=stn,UNIT=un)

BACKSPACE(UNIT=2*IN+2)

BACKSPACE(IOSTAT=IOS,ERR=99999,UNIT=2*IN-10)
```

Invalid BACKSPACE Statements:

`BACKSPACE UNIT=un`	UNIT= is not allowed without the parentheses.
`BACKSPACE un,ERR=stn`	Parentheses must be specified.
`BACKSPACE (ERR=stn,un)`	UNIT= must be specified.

When the BACKSPACE statement is encountered, the unit specified by un must be connected to an external file for SEQUENTIAL access. (See VS FORTRAN Application Programming: Guide.) If the unit is not connected, an error is detected.

The external file connected to the unit un must exist; otherwise, an error is detected. (The existence of a file can be determined with the INQUIRE statement. exs must have the value true. see "INQUIRE Statement" on page 125.)

A BACKSPACE statement positions an external file to the beginning of the preceding record. If there is no preceding record, the

BACKSPACE statement has no effect. The BACKSPACE statement must not be used with external files using list-directed formatting.

IBM EXTENSION

The BACKSPACE statement must not be used with external files written using NAMELIST. If it is used, the result is unpredictable.

An external file can be extended if the execution of an ENDFILE statement or the detection of an end-of-file is immediately followed by the execution of a BACKSPACE and a WRITE statement on this file. (See "READ Statement—Formatted with Sequential Access" on page 150.)

The BACKSPACE statement may be used with asynchronous READ and WRITE statements provided that any input or output operation on the file has been completed by the execution of a WAIT statement. A WAIT statement is not required to complete the BACKSPACE operation.

END OF IBM EXTENSION

Transfer is made to the statement number specified by the ERR parameter if an error is detected. If IOSTAT=ios is specified, a positive integer value is assigned to ios when an error is detected. Execution continues with the statement number specified by the ERR parameter (if present) or with the next statement if the ERR parameter is not specified. If the ERR parameter and the IOSTAT parameter are both omitted, program execution is terminated when an error is detected.

BLOCK DATA STATEMENT

The BLOCK DATA statement names a block of data.

Syntax

```
BLOCK DATA [name]
```

name
is the name of the block data subprogram. This name is optional. It must not be the same as the name of another subprogram, a main program, or common block name in the executable program. There can only be one unnamed block data subprogram in an executable program.

To initialize variables in a named common block, a separate subprogram must be written. This separate subprogram contains only the BLOCK DATA, IMPLICIT, PARAMETER, DATA, COMMON, DIMENSION, SAVE, EQUIVALENCE, and END statements, comment lines, and explicit type specification statements associated with the data being defined. This subprogram is not called; its presence provides initial data values for named common blocks. Data may not be initialized in unnamed common blocks.

The BLOCK DATA statement must appear only as the first statement in the subprogram. Statements that provide initial values for data items cannot precede the COMMON statements that define those data items.

Any main program or subprogram using a named common block must contain a COMMON statement defining that block. If initial values are to be assigned, a block data subprogram is necessary.

A particular common block may not be initialized in more than one block data subprogram.

Entities not in a named common block must not be initialized and must not appear in a DIMENSION, EQUIVALENCE, or type statement in a block data subprogram.

All elements of a named common block must be listed in the COMMON statement, even though they are not all initialized. For example, the variable A in the COMMON statement in the following block data subprogram does not appear in the DATA statement.

Example 1:

```
BLOCK DATA
COMMON /ELN/C,A,B
COMPLEX C
DATA C/(2.4,3.769)/,B/1.2/
END
```

Data may be entered into more than one common block in a single block data subprogram.

Example 2:

```
BLOCK DATA VALUE1
COMMON/ELN/C,A,B/RMG/Z,Y
COMPLEX C
DOUBLE PRECISION Z
DATA C/(2.4,3.769)/,B/1.2/,Z/7.64980825D0/
END
```

As a result of this example, in BLOCK DATA named VALUE1,

```
COMMON/ELN/C,A,B
```

will have the complex variable C real part initialized to 2.4 and the imaginary part initialized to 3.769. The variable A will not be initialized and B will be initialized to 1.2.

```
COMMON/RMG/Z,Y
```

will have the double precision variable Z initialized with the double precision constant 7.64980825 and Y will not be initialized.

BLOCK IF STATEMENT

See "IF Statements" on page 117.

CALL STATEMENT

The CALL statement:

- Transfers control to a subroutine subprogram
- Evaluates actual arguments that are expressions
- Associates actual arguments with dummy arguments

Syntax

```
CALL name [ ( [arg1 [,arg2] [,arg3] ... ] ) ]
```

name
is the name of a subroutine subprogram or an entry point. This name may be a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement.

arg
is an actual argument that is being supplied to the subroutine subprogram. The argument may be a variable, array element or array name, a constant, an arithmetic, logical, or character expression, a function or subroutine name, or an asterisk (*) followed by the statement number of an executable statement that appears in the same program unit as the CALL statement.

If no actual argument is specified, the parentheses may be omitted.

The CALL statement transfers control to the subroutine subprogram and replaces the dummy variables with the values of the actual arguments that appear in the CALL statement.

The CALL statement can be used in a main program, a function subprogram, or a subroutine subprogram, but a subprogram must not refer to itself directly or indirectly and must not refer to the main program. A main program cannot call itself.

If name is a dummy argument in a subprogram containing CALL name, this CALL statement can be executed only if the subprogram is given control at one of its entry points where name appears in the list of dummy arguments. (See "EXTERNAL Statement" on page 89.)

Valid CALL Statements:

```
CALL SZ0001
CALL SZ0001( )
CALL S19001(CVAR40)
CALL TEST2(TF1,KF2,JIF3)
CALL SUB1(COM2+3*COM3-7,VAL2*VAL3/.6,.TRUE.)
CALL SUB2(A,B,*10,*20,*30)
CALL B('A',0,1,R)
```

CHARACTER TYPE STATEMENT

See "Explicit Type Statement" on page 85.

CLOSE STATEMENT

A CLOSE statement disconnects an external file from an input or output unit.

Syntax

```
CLOSE ( [UNIT=]un [,ERR=stn] [,STATUS=sta] [,IOSTAT=ios] )
```

UNIT=un
un is the reference to the number of an I/O unit. It can optionally be preceded by UNIT=. It can be an integer or real arithmetic expression. Its value (after conversion to integer of length 4, if necessary) must be zero or positive; otherwise, an error is detected.

If UNIT= is not specified, un must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=stn
is optional. stn is a statement number. If an error occurs in the execution of the CLOSE statement, control is transferred to the statement labeled stn. That statement must be executable and must be in the same program unit as the CLOSE statement. If ERR=stn is omitted, execution halts when an error is detected.

STATUS=sta
is optional. sta is a character expression whose value (when any trailing blanks are removed) must be KEEP or DELETE. sta determines the disposition of the file that is connected to the specified unit.

IOSTAT=ios
is optional. ios is an integer variable or an integer array element of length 4. Its value is set positive if an error is detected; it is set to zero if no error is detected. VSAM return and reason codes are placed in IOSTAT.

Each of the parameters of the CLOSE statement may appear only once. The unit specifier (un) must appear. All value assignments are made according to the rules for assignment statements.

Execution of a CLOSE statement that refers to a unit may occur in any program unit of an executable program and need not occur in the same program unit as the execution of an OPEN statement referring to that unit. When the CLOSE statement is encountered, the unit specified by un may or may not be connected to a file. If the unit is connected, the file may or may not exist.

If KEEP is specified for a file that exists, the file continues to exist after the execution of the CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of the CLOSE statement. If DELETE is specified, the file is deleted.

If STATUS is omitted, the assumed value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case the assumed value is DELETE. (The STATUS parameter affects only the internal VS FORTRAN status. The external status is set by the JCL or other system environment and will not be overridden.)

After a **unit** has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program to the same file or a different file.

After a **file** has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable

program to the same unit or a different unit provided that the file still exists. (See "OPEN Statement" on page 134.)

When execution ends normally, all units that are connected are closed. Each unit is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with status DELETE.

Example 1:

Assume that the type of the following variables has been specified as follows:

Variable Names	Type	Length
IN,IACT,Z	INTEGER	4
DELETE	CHARACTER	6

and that

```
DELETE = 'DELETE'
```

The following statements are valid:

```
CLOSE(6+IN)
CLOSE(Z*IN+2)
CLOSE(Z*IN+3,STATUS=DELETE)
CLOSE(IOSTAT=IACT,ERR=99999,STATUS='KE'//'EP ',UNIT=0)
```

Example 2:

```
STATUS='KEEP'
DELETE=STATUS
CLOSE(UNIT=6,STATUS=DELETE)
CLOSE(UNIT=6,STATUS=STATUS)
CLOSE(UNIT=6,STATUS='KEEP')
```

Each of these CLOSE statements should execute the same way and give a status of KEEP.

COMMENTS

Comments provide documentation for a program. They can be entered in either fixed form or free form.

Fixed-Form Input

Fixed-form comments have the following attributes:

- A "C" or an asterisk (*) may appear in column 1 or all blanks may appear in columns 1 to 72.
- A comment may appear anywhere before the END statement.

IBM EXTENSION

Free-Form Input

Free-form comments have the following attributes:

- Any line that does not follow a continued line and that has the quotation mark (") character as its first character is considered a comment.
- A comment line cannot be continued.

END OF IBM EXTENSION

COMMON STATEMENT

The COMMON statement makes it possible for two or more program units to share storage and to specify the names of variables and arrays that are to occupy the area.

Syntax

```
COMMON [/[name1]/] list1[ [,] /[namen]/ listn ] ...
```

name
is an optional common block name. These names must always be enclosed in slashes. They cannot be the same as names used in PROGRAM, SUBROUTINE, FUNCTION, ENTRY, or BLOCK DATA statements.

The form // (with no characters except possibly blanks between the slashes) denotes blank common. If name1 denotes blank common, the first two slashes are optional.

The comma preceding the common block name designator /name/ is optional.

list
is a list of variable names or array names that are not dummy arguments. If a variable name is also a function name, subroutine name, or entry name, it must not appear in the list. If the list contains an array name, dimensions may also be declared for that array. (See "DIMENSION Statement" on page 71.)

A given common block name may appear more than once in a COMMON statement, or in more than one COMMON statement in a program unit.

Blank and named common entries appearing in COMMON statements are cumulative throughout the program unit. Consider the following two COMMON statements:

```
COMMON A, B, C /R/ D, E /S/ F

COMMON G, H /S/ I, J /R/R//W
```

These two statements have the same effect as the single statement:

```
COMMON A, B, C, G, H, W /R/ D, E, R /S/ F, I, J
```

If a character variable or character array is in a common block, all the other variables and arrays in that common block must be of type character.

Although the entries in a COMMON statement can contain dimension information, object-time dimensions may never be used.

The length of a blank common can be extended by using an EQUIVALENCE statement, but only by adding beyond the last entry.

A common block resides in a fixed location in storage during the execution of a program. Because of this, all program units of this program refer to data at that location as defined in the COMMON statements in each program unit.

In the following example, the complex variable, CV, and the real array, RV, refer to the same storage locations.

The statement: RV(2) = 1.2 will assign the value of 1.2 to the imaginary part of CV.

```
Main Program                    Subroutine

COMMON CV                       SUBROUTINE SUB
COMPLEX*8 CV                    COMMON RV(2)
  .                               .
  .                               .
  .                               .
CALL SUB                        RV(2) = 1.2
  .                               .
  .                               .
  .                               .
STOP                            RETURN
END                             END
```

Blank and Named Common

Variables and arrays may be placed in separate common blocks by giving distinct common block names (name). Those blocks that have the same name occupy the same storage area. The name cannot be the same as the main program name, subprogram name, or entry name. The variables and arrays of a common block must all be of type character or all noncharacter in all the program units that refer to the common block.

Naming these separate blocks permits a calling program to share one common block with one subprogram and another common block with another subprogram. It also makes it easier to document the program.

The differences between blank and named common are:

- There is only one blank common in an executable program, and it has no name.

  There may be many named commons, each with its own name.

- Blank common may have different lengths in different program units.

  Each program unit that uses a named common must define it to be of the same length.

- Variables and array elements in blank common cannot be assigned initial values.

  Variables and array elements in named common may be assigned initial values by DATA statements in a block data subprogram.

IBM EXTENSION

Variables and array elements in named common may be assigned initial values by explicit type specification statements in a block data subprogram.

END OF IBM EXTENSION

Variables that are to be placed in named common are preceded by the common block name enclosed in slashes. For example, the variables A, B, and C are placed in the named common, HOLD, by the following statement:

```
COMMON /HOLD/ A,B,C
```

In a COMMON statement, blank common is distinguished from named common by placing two consecutive slashes before the variables (or, if the variables appear at the beginning of the COMMON statement, by omitting any common block name). For example,

```
COMMON A, B, C /ITEMS/ X, Y, Z / / D, E, F
```

COMMON

The variables A, B, C, D, E, and F are placed in blank common in that order; the variables X, Y, and Z are placed in the named common ITEMS.

COMPLEX TYPE STATEMENT

See "Explicit Type Statement" on page 85.

COMPUTED GO TO STATEMENT

See "GO TO Statements" on page 115.

CONTINUE STATEMENT

The CONTINUE statement is an executable control statement that takes no action. It can be used to designate the end of a DO loop, or to label a position in a program.

Syntax

```
CONTINUE
```

CONTINUE
is a statement that may be placed anywhere in the source program (where an executable statement may appear) without affecting the sequence of execution. It may be used as the last statement in the range of a DO loop in order to avoid ending the DO loop with an unconditional or assigned GO TO, block IF, ELSE IF, ELSE, ENDIF, STOP, RETURN, END, arithmetic IF, another DO statement, or a logical IF statement containing an unconditional or assigned GO TO, or a STOP, RETURN, or arithmetic IF statement.

DATA STATEMENT

The DATA statement defines initial values of variables, array elements, arrays, and substrings.

Syntax

```
DATA list1 /clist1/ [ [,] list2 /clist2/ ] ...
```

list
is a list of variables, array elements, arrays or substrings, and implied DO lists. The comma preceding list2...listn is optional.

Subscript and substring expressions used in each list can contain only integer constants or names of integer constants. (An exception is described under "Implied DO in a DATA Statement" on page 74.)

clist
is a list of constants or the names of constants. Integer and real constants may optionally be signed. Any of these constants may be preceded by r*, where r is a nonzero unsigned integer constant or the name of such a constant. When the form r* appears before a constant, it indicates that the constant is to be repeated r times.

A DATA initialization statement is not executable. The DATA statement cannot precede a PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA, IMPLICIT, PARAMETER, or an explicit type statement. Otherwise, a DATA statement can appear anywhere in the program.

There must be a one-to-one correspondence between the total number of elements specified or implied by the list list and the total number of constants specified by the corresponding list clist after application of any replication factors, r.

Integer, real, and complex variables or array elements must be initialized with integer, real, or complex constants; conversions take place according to the arithmetic assignment rules, if necessary.

IBM EXTENSION

A hexadecimal constant can be used to initialize any arithmetic or logical type of variable or array element.

If a hexadecimal constant initializes a complex data type, one constant is used that initializes both the real and the imaginary parts and the constant is not enclosed in parentheses. If the constant is smaller than the length (in bytes) of the entire complex entity, zeros are added on the left. If the constant is larger, the leftmost hexadecimal digits are truncated.

A logical variable or logical array may be initialized with T instead of .TRUE. and F instead of .FALSE..

END OF IBM EXTENSION

Character items can be initialized by character data only. Each character constant initializes exactly one variable, one array element, or one substring. If a character constant contains more characters than the item it initializes, the additional rightmost characters in the constant are ignored. If a character constant contains fewer characters than the item it initializes, the additional rightmost characters in the item are initialized with blank characters. (Each character represents one byte of storage.)

A variable or array element defined with an initial value may not be in blank common and may not be assigned an initial value more than once. If the variable or array element is in a named common block, it may be initially defined only in a block data subprogram. For purposes of this constraint, entities that are associated with each other through COMMON or EQUIVALENCE statements are considered as the same entity.

Valid DATA Statements:

```
DATA A, B, C/5.0,6.1,7.3/,D/25*1.0,25*2.0/,E/5.1/

DATA F/5*1.0/, G/9*2.0/, L/4*.TRUE./, C/'FOUR'/

DATA CC(1)(1:2)/'AB'/,CC(1)(3:4)/'CD'/
```

IBM EXTENSION

DEBUG STATEMENT

The DEBUG statement sets the conditions for operation of the debug facility and designates debugging operations that apply to the entire program unit (such as subscript checking).

Syntax

```
DEBUG option,..., option
```

An option may be any of the following:

UNIT (un)
un is an integer constant that represents a unit number. All debugging output is placed in this file called the debug output file. If this option is not specified, any debugging output is placed in the installation-defined output file. All unit definitions within an executable program must refer to the same unit.

SUBCHK (a1, a2,..., an)
a is an array name. The validity of the subscripts used with the named arrays is checked by comparing the subscript combination with the size of the array. If the subscript value exceeds the size of the array, a message is placed in the debug file. Program execution continues, using the incorrect subscript. If the list of array names is omitted, all arrays in the program are checked for valid subscript usage. If the entire option is omitted, no arrays are checked for valid subscripts.

TRACE
This option must be in the DEBUG specification statement of each program or subprogram for which tracing is desired. If this option is omitted, there can be no display of program flow by statement number within this program. Even when this option is used, a TRACE ON statement must appear in the first debug packet in which tracing is desired.

INIT (i1, i2,..., in)
i is the name of a variable or an array that is to be displayed in the debug output file only when the variable or the array elements are assigned a value. If i is a variable name, the name and value are displayed whenever the variable is assigned a new value in either an assignment, a READ or an ASSIGN statement. If i is an array name, the array element is displayed. If the list of names is omitted, a display occurs whenever the value of a variable or an array element is assigned a value. If the entire option is omitted, no display occurs when values are assigned.

SUBTRACE
This option specifies that the name of this subprogram is to be displayed whenever it is entered. The message RETURN is to be displayed whenever execution of the subprogram is completed.

The options in a DEBUG statement may be given in any order and they must be separated by commas.

All debugging statements must precede the first statement of the program being debugged. The required statement sequence is:

1. DEBUG statement
2. Debug packets

3. END DEBUG statement

4. First of the source program statements of a program unit to be debugged

A debug packet begins with an AT statement and ends when either another AT statement or an END DEBUG statement is encountered.

Debug statements are written in either fixed form or free form and follow the same rules as other VS FORTRAN statements.

In addition to the VS FORTRAN language statements, the following debug statements are allowed:

```
TRACE ON
TRACE OFF
DISPLAY
```

All VS FORTRAN statements are allowed in a debug packet except as listed in "Considerations when Using DEBUG."

Considerations when Using DEBUG

The following precautions must be taken when setting up a debug packet:

- Any DO loops, block IF, ELSE IF, or ELSE statements initiated within a debug packet must be wholly contained within that packet.
- Statement numbers within a debug packet must be unique. They must be different from statement numbers within other debug packets and within the program being debugged.
- An error in a program should not be corrected with a debug packet; when the debug packet is removed, the error remains in the program.
- No specification statements can appear in a debug packet; nor can any of the following statements:

  ```
  BLOCK DATA
  ENTRY
  FUNCTION
  PROGRAM
  statement function
  SUBROUTINE
  ```

- The program being debugged must not transfer control to any statement number defined in a debug packet; however, control may be returned to any point in the program being debugged from a packet. In addition, no debug packet may refer to a label defined in another debug packet. A debug packet may contain a RETURN, STOP, or CALL statement.

END OF IBM EXTENSION

DEBUG Examples:

Example 1:

```
      DEBUG UNIT(6)
      AT 11
      WRITE(6,21)A,B,C
   21 FORMAT(1X,'A=',I10,'B=',I10,'C=',I10)
      END DEBUG
        .
        .
        .
      INTEGER A,B,C
        .
        .
        .
   10 B=A* SQRT(FLOAT(C))
   11 IF(B)40,50,60
        .
        .
        .
```

The values of A, B, and C are to be examined as they were at the completion of the arithmetic operation in statement 10. Therefore, the statement number specified in the AT statement is 11. The values of A, B, and C are written to the file connected to unit 6.

Example 2:

```
      DEBUG TRACE, UNIT(6)
      AT 10
      TRACE ON
      AT 25
      TRACE OFF
      AT 35
      DISPLAY C
      TRACE ON
      END DEBUG
        .
        .
        .
   10 A=2.0
   15 L=1
   20 B=A+1.5
   25 DO 30 I=1,5
        .
        .
        .
   30 CONTINUE
   35 C=B+3.415
   40 D=C**2
   45 CALL SUB1(D,L,R)
        .
        .
        .
```

When statement 10 is encountered, tracing begins, as specified by the TRACE ON statement in the first debug packet. When statement 25 is encountered, tracing stops, as specified by the TRACE OFF statement in the second debug packet. When statement 35 is encountered, tracing begins again and the value of C is written to the debug output file, as specified in the third debug packet.

DIMENSION STATEMENT

The DIMENSION statement specifies the name and dimensions of an array.

Syntax

```
DIMENSION a1(dim1) [, a2(dim2) ] ...
```

a
is an array name.

dim
is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array in the form:

```
e1:e2
  or
e2
```

where:

e1
is the lower dimension bound. It is optional. If e1 (with its following colon) is not specified, its value is assumed to be 1.

e2
is the upper dimension bound and must always be specified.

(See "Size and Type Declaration of an Array" on page 22 for rules about dimension bounds.)

Each a in a DIMENSION statement declares that a is an array in that program unit. Array names and their bounds may also be declared in COMMON statements and in type statements. Only one declaration of the array name (a) as an array is permitted in a program unit.

Valid DIMENSION Statements:

```
DIMENSION A(10), ARRAY(5,5,5), LIST(10,100)

DIMENSION A(1:10), ARRAY(1:5,1:5,1:5), LIST(1:10,1:100)

DIMENSION B(0:24), C(-4:2), DATA(0:9,-5:4,10)

DIMENSION G(I:J,M:N)

DIMENSION ARRAY (M*N:I*J)

DIMENSION ARRAY (M*N:I*J,*)
```

IBM EXTENSION

DISPLAY STATEMENT

The DISPLAY statement displays data in NAMELIST output format. It may appear anywhere within a debug packet.

Syntax

```
DISPLAY list
```

list
is a list of variable or array names separated by commas.

The DISPLAY statement eliminates the need for FORMAT or NAMELIST and WRITE statements to display the results of a debugging operation. The data is placed in the debug output file.

The effect of a DISPLAY list statement is the same as the following source language statements:

```
NAMELIST /name/list

WRITE (un, name)
```

where name is the same in both statements.

Array elements, dummy arguments, and substring references may not appear in the list.

For examples and explanations of the DISPLAY statement, see "DEBUG Statement" on page 68.

END OF IBM EXTENSION

DO STATEMENT

The DO statement indicates that the statements that physically follow it, up to and including a specified statement, are to be executed. These statements are called the "range of the DO" or a "DO-loop."

Syntax

	End of Range		DO Variable		Initial Value	Test Value	Increment
DO	stn	[,]	i	=	m1,	m2	[,m3]

stn
is the number of an executable statement appearing after the DO statement in the program unit containing the DO. The comma after stn is optional.

i
is an integer, real, or double precision variable (not an array element) called the DO variable.

m1, m2, and m3,
are integer, real, or double precision arithmetic expressions. The values of the expressions m1, m2 and m3 are converted to the type of the DO variable i, if necessary. m3 is optional and cannot have a value of zero; if it is omitted, its value is assumed to be 1, and the preceding comma must be omitted.

The statements in the range of the DO are executed only if:

m1 is less than or equal to m2, and m3 is greater than 0
or
m1 is greater than or equal to m2, and m3 is less than 0

If one of the above relationships between m1, m2, and m3 is true, the first time the statements in the range of the DO are executed, i is initialized to the value of m1; on each succeeding iteration, i is increased by the value of m3. The number of iterations that can be executed, also called iteration count, is the value of:

MAX (INT((m2 - m1 + m3) / m3), 0).

The first time i exceeds m2 at the end of the iteration, control passes to the statement following the statement numbered stn. Upon completion of the DO, the DO variable i contains the last value that exceeded m2.

If one of the above relationship is not true, execution continues with the statement following the last statement of the range of the DO or the outer DO if the statement numbered stn is shared by more than one DO. (See "IF Statements" on page 117.)

Valid DO Statements:

```
DO 40, INT=1,4,1

DO 20, VAR=START,END,INC
```

For examples (with explanations) of DO statements (including nesting), see VS FORTRAN Application Programming: Guide.

Implied DO in a DATA Statement

The form of an implied DO list in a DATA statement is:

Syntax

```
(dlist, i = m1, m2 [, m3] )
```

where:

dlist
is a list of array element names and implied DO lists.

i
is the name of an integer variable called the implied DO variable.

m1, m2, and m3
are each an integer constant or name of an integer constant, or an expression containing only integer constants or names of integer constants. The expression may contain implied DO variables of other surrounding implied DO lists that have this implied DO list within their ranges (dlist). m3 is optional and, if omitted, it is assumed to be 1, and the preceding comma must be omitted.

The range of an implied DO list is dlist. An iteration count is established from m1, m2, and m3 exactly as for a DO-loop except that the iteration count must be positive.

Upon completion of the implied DO, the implied DO variable is undefined and may not be used until assigned a value in a DATA statement, assignment statement, or READ statement.

Each subscript expression in dlist must be an integer constant or an expression containing only integer constants or names of integer constants. The expression may contain implied DO variables of implied DO lists that have the subscript expression within their ranges.

Valid Implied DO Statement:

```
DATA ((X(J,I),I=1,J),J=1,5)/15*0./
```

Implied DO in an Input/Output Statement

If an implied DO appears in the list parameter of an input/output statement, the items specified by the implied DO are transmitted to or from the file. The implied DO list in an input/output statement is of the form:

```
(dlist, i = m1, m2 [, m3] )
```

where:

dlist
is an input/output list.

i
is the name of an integer, real, or double precision variable (not an array element) called the DO variable.

m1, m2, and m3
are integer, real, or double precision arithmetic expressions. The values of the expressions m1, m2, and m3 are converted to the type of the DO variable i, if necessary. m3 is optional and cannot have a value of zero; if it is omitted, its value is assumed to be 1, and the preceding comma must be omitted.

In an input statement, the DO-variable i, or an associated entity, must not appear as an input list item in dlist. When an implied-DO list appears in an input/output list, the list items in dlist are specified once for each iteration of the implied DO list with appropriate substitution of values for any occurrence of the DO-variable i.

For example, assume that A is a variable and that B, C, and D are one-dimensional arrays, each containing 20 elements. Then the statement:

```
READ (UNIT=5)A,B,(C(I),I=1,4),D(4)
```

reads one value into A, the next 20 values into B, and the next 4 values into the first four elements of the array C, and the next value into the fourth element of D.

Or the statement:

```
WRITE (UNIT=6)A,B,(C(I),I=1,4),D(4)
```

writes one value from A, the next 20 values from B, and the next 4 values from the first four elements of the array C, and the next value from the fourth element of D.

If the subscript (I) were not included with the array C, the entire array would be transferred four times.

Implied DO's can be nested, if required. For example, to read an element into array B after values are read into each row of a 10x20 array A, the following input statement would be written:

```
READ (UNIT=5)((A(I,J),J=1,20),B(I),I=1,10)
```

Or to write an element from array B after values are written into each row of a 10x20 array A, the following output statement would be written:

```
WRITE (UNIT=6)((A(I,J),J=1,20),B(I),I=1,10)
```

The order of the names in the list specifies the order in which the data is transferred.

DOUBLE PRECISION TYPE STATEMENT

See "Explicit Type Statement" on page 85.

EJECT

IBM EXTENSION

EJECT STATEMENT

EJECT is a compiler directive. It starts a new full page of the source listing. The EJECT statement should not be continued.

Syntax

```
EJECT
```

END OF IBM EXTENSION

ELSE STATEMENT

See "IF Statements" on page 117.

ELSE IF STATEMENT

See "IF Statements" on page 117.

END STATEMENT

The END statement defines a program unit. That is, it terminates a main program, or a function, subroutine, or block data subprogram.

Syntax

```
END
```

The END statement may be numbered. It may not be continued and no other statement in the program unit may have an initial line that appears to be an END statement. The END statement terminates program execution if it is executed in the main program. If executed in a subprogram, it has the effect of a RETURN statement.

Execution of an END statement terminates the association between the dummy arguments of the subprogram and the current actual arguments. All entities within the subprogram become undefined except:

- Entities specified in SAVE statements (see "SAVE Statement" on page 168)
- Entities in blank common.
- Initially defined entities that have neither been redefined nor become undefined.
- Entities in named common blocks that appear in the subprogram and appear in at least one other program unit that is referring, either directly or indirectly, to that subprogram. The entities in a named common block may become undefined by execution of a RETURN or END statement in another program unit.

All variables that are assigned a statement number with the ASSIGN statement become undefined regardless of whether the variable is in common or specified in a SAVE statement.

END Statement in a Function Subprogram

All function subprograms must end with an END statement. They may also contain RETURN statements. The END statement specifies the physical end of the subprogram.

A subprogram must not be referred to twice during the execution of an executable program without the intervening execution of a RETURN or END statement in that subprogram.

END Statement in a Subroutine Subprogram

All subroutine subprograms must end with an END statement. They may also contain RETURN statements. The END statement specifies the physical end of the subprogram. If the END statement is reached during execution of the subroutine subprogram, it is executed as a RETURN statement.

IBM EXTENSION

END DEBUG STATEMENT

The END DEBUG statement terminates the last debug packet for the program.

Syntax

```
END DEBUG
```

END DEBUG is placed after the other debug statements and just before the first statement of the program being debugged. Only one END DEBUG statement is allowed in a program unit.

For examples of debug packets and the END DEBUG statement, see "DEBUG Statement" on page 68.

END OF IBM EXTENSION

ENDFILE STATEMENT

The ENDFILE statement writes an end-of-file record on a sequentially accessed external file.

Syntax

```
ENDFILE un

ENDFILE ( [UNIT=]un [, ERR=stn] [, IOSTAT=ios] )
```

UNIT=un
is the reference to the number of an I/O unit. un can optionally be preceded by UNIT= if the second form of the statement is used. It can be an integer or real arithmetic expression. Its value (after conversion to integer of length 4, if necessary) must be zero or positive; otherwise, an error is detected.

ERR=stn
is optional. stn is a statement number. If an error occurs in the execution of the ENDFILE statement, control is transferred to the statement labeled stn. That statement must be executable and must be in the same program unit as the ENDFILE statement. If ERR=stn is omitted, execution halts when an error is detected.

IOSTAT=ios
is optional. ios is an integer variable or an integer array element of length 4. Its value is set positive if an error is detected; it is set to zero if no error is detected. VSAM return and reason codes are placed in ios.

If UNIT= is specified, UNIT, ERR, and IOSTAT can appear in any order; otherwise, un must appear first.

Valid ENDFILE Statements:

```
ENDFILE un

ENDFILE (un,ERR=stn)

ENDFILE (UNIT=un,ERR=stn)

ENDFILE (ERR=stn,UNIT=un)
```

Invalid ENDFILE Statements:

`ENDFILE UNIT=un`	UNIT= is not allowed outside parentheses.
`ENDFILE un,ERR=stn`	Parentheses must be specified.
`ENDFILE (ERR=stn,un)`	UNIT= must be specified or un must be first in the list.

When the ENDFILE statement is encountered, the unit specified by un must be connected to an external file with SEQUENTIAL access. (See VS FORTRAN Application Programming: Guide for an example.) If the unit is not connected, an error is detected.

After successful execution of the ENDFILE statement, the external file connected to the unit specified by un is created if it does not already exist.

ENDFILE

IBM EXTENSION

Use of ENDFILE with asynchronous READ and WRITE statements is allowed provided that any input or output operation on the file has been allowed to complete by the execution of a WAIT statement. A WAIT statement is not required to complete the ENDFILE operation.

Transfer is made to the statement specified by the ERR= if an error is detected. If IOSTAT=ios is specified, a positive integer value is assigned to ios when an error is detected. Then execution continues with the statement specified with the ERR parameter, if present, or with the next statement if ERR is not specified. If the ERR parameter and the IOSTAT parameter are both omitted, program execution is terminated when an error is detected.

Multiple file data sets are permitted in VS FORTRAN. Therefore, after execution of an ENDFILE, additional data may be transferred to the subsequent files.

END OF IBM EXTENSION

END IF STATEMENT

See "IF Statements" on page 117.

ENTRY STATEMENT

The ENTRY statement names the place in a subroutine or function subprogram that can be used in a CALL statement or as a function reference.

The normal entry into a subroutine subprogram from the calling program is made by a CALL statement that refers to the subprogram name. The normal entry into a function subprogram is made by a function reference in an arithmetic, character, or logical expression. Entry is made at the first executable statement following the SUBROUTINE or FUNCTION statement.

It is also possible to enter a subprogram by a CALL statement (for a subroutine subprogram) or a function reference (for a function subprogram) that refers to an ENTRY statement in the subprogram. Entry is made at the first executable statement following the ENTRY statement.

Syntax

```
ENTRY name [ ( [ arg1 [, arg2 ] ... ] ) ]
```

name
is the name of an entry point in a subroutine or function subprogram. If ENTRY appears in a subroutine subprogram, name is a subroutine name. If ENTRY appears in a function subprogram, name is a function name.

arg
is an optional dummy argument corresponding to an actual argument in a CALL statement or in a function reference. See "Subprogram Statements" on page 43. If no arg is specified, the parentheses are optional.

arg may be a variable name, array name, or dummy procedure name or an asterisk. An asterisk is permitted only in an ENTRY statement in a subroutine subprogram.

The ENTRY statement cannot appear in a main program.

A function subprogram must not refer to itself or any of its entry points either directly or indirectly.

ENTRY statements are nonexecutable and do not affect control sequencing during execution of a subprogram. They can appear anywhere after a FUNCTION or SUBROUTINE statement except that an ENTRY statement must not appear between a block IF statement and its matching END IF statement or between a DO statement and the terminal statement of its range.

Note: ENTRY statements can appear before the IMPLICIT or PARAMETER statements. The appearance of an ENTRY statement does not alter the rule that statement functions must precede the first executable statement.

Within a function or subroutine subprogram, an entry name must not appear as a dummy argument of a FUNCTION, SUBROUTINE, or ENTRY statement and it must not appear in an EXTERNAL statement.

If information for an object-time dimension array is passed in a reference to an ENTRY statement, the array name and all its dimension parameters (except any that are in a common area) must appear in the argument list of the ENTRY statement. See "Size and Type Declaration of an Array" on page 22.

In a function subprogram, the type of the function name and entry name are determined (in order of decreasing priority) by:

1. An explicit type statement
2. An IMPLICIT statement
3. Predefined convention

In function subprograms, an entry name must not appear preceding the entry statement except in a type statement.

If any entry name in a function subprogram or the name of the function subprogram is of type character, all entry names of the function subprogram must be of type character with the same length. The CHARACTER type statement or IMPLICIT statement can be used to specify the type and length of the entry point name. The length specification is restricted to the forms permitted in the FUNCTION statement.

The types of these variables (that is, the function name and entry names) can be different only if the type is not character; the variables are treated as if they were equivalenced. After one of these variables is assigned a value in the subprogram, any others of different type become indeterminate in value.

In a function subprogram, either the function name or one of the entry names must be assigned a value.

Upon exit from a function subprogram, the value returned is the value last assigned to the function name or any entry name. It is returned as though it were assigned to the name in the current function reference. If the last value is assigned to a different entry name, and that entry name differs in type from the name in the current function reference, the value of the function is undefined.

Entry names in a subroutine subprogram do not have a type; explicit typing is not allowed.

Valid ENTRY Statements:

```
ENTRY ENT(T)

ENTRY SUB2 (T,*,*)

ENTRY SUB3 (*,*)
```

Actual Arguments in an ENTRY Statement

Entry into a function subprogram associates actual arguments with the dummy arguments of the referenced ENTRY statement. Thus, all appearances of these arguments in the whole subprogram become associated with actual arguments.

See "Actual Arguments in a Subroutine Subprogram" on page 173 and "Actual Arguments in a Function Subprogram" on page 113.

Dummy Arguments in an ENTRY Statement

The dummy arguments in the ENTRY statement need not agree in order, type, or number with the dummy arguments in the SUBROUTINE or FUNCTION statement or any other ENTRY statement in the same subprogram. However, the actual arguments for each CALL or function reference must agree in order, type, and number with the dummy arguments in the SUBROUTINE, FUNCTION, or ENTRY statement to which it refers.

Any dummy argument of an ENTRY statement must not be in an executable statement preceding the ENTRY statement unless it has already appeared as a dummy argument in an ENTRY, SUBROUTINE, or FUNCTION statement prior to the executable statement.

If an ENTRY dummy argument is used as an adjustable array name, the array name and all its dimensions (except those in COMMON) must be in the same dummy argument list.

Dummy arguments can be variables, arrays, dummy procedure names, or asterisks. The asterisk is allowed only in an ENTRY statement in a subroutine subprogram and indicates an alternate return specifier.

A dummy argument must not appear in the expression of a statement function definition unless the name is also a dummy argument to the statement function, or is in a FUNCTION or SUBROUTINE statement, or is in an ENTRY prior to the statement function definition.

A dummy argument used in an executable statement is allowed only if that dummy argument appears in the argument list of the FUNCTION, SUBROUTINE, or ENTRY statement by which the subprogram was entered.

See "Dummy Arguments in a Subroutine Subprogram" on page 174 and "Dummy Arguments in a Function Subprogram" on page 113.

EQUIVALENCE STATEMENT

The EQUIVALENCE statement permits the sharing of data storage within a single program unit.

Syntax

```
EQUIVALENCE (list1) [, (list2) ] ...
```

list
is a list of variable, array, array element, or character substring names. Names of dummy arguments of an external procedure in a subprogram must not appear in the list. Each pair of parentheses must contain at least two names.

The number of subscript quantities of array elements must be equal to the number of dimensions of the array. If an array name is used without a subscript in the EQUIVALENCE statement, it is interpreted as a reference to the first element of the array.

An array element refers to a position in the array in the same manner as it does in an assignment statement (that is, the array subscript specifies a position relative to the first element of each dimension of the array).

The subscripts and substring information may be integer expressions containing only integer constants, or names of integer constants. They must not contain variables, array elements, or function references.

All the named data within a single set of parentheses share the same storage location. When the logic of the program permits it, the EQUIVALENCE statement can reduce the number of bytes used by sharing two or more variables of the same type or different noncharacter types. Character type variables and character type array elements can only be equivalenced with other character type variables, character type array elements, or portions of them. The length of the equivalenced entities can be different. Equivalence between variables implies storage sharing.

Mathematical equivalence of variables or array elements is implied only when they are of the same noncharacter type, when they share exactly the same storage, and when the value assigned to the storage is of that type.

Because arrays are stored in a predetermined order, equivalencing two elements of two different arrays implicitly equivalences other elements of the two arrays. The EQUIVALENCE statement must not contradict itself or any previously established equivalences.

Two variables in one common block or in two different common blocks cannot be made equivalent. However, a variable in a program unit can be made equivalent to a variable in a common block. If the variable that is equivalenced to a variable in the common block is an element of an array, the implicit equivalencing of the rest of the elements of the array can extend the size of the common block. The size of the common block cannot be extended so that elements are added ahead of the beginning of the established common block.

Valid EQUIVALENCE Statements:

```
EQUIVALENCE (C(1), A(1)), (C(50,50), B(1))

EQUIVALENCE (A, B(1), C(5,3)), (D(5,10,2), E)

EQUIVALENCE (B,D(1))
```

EXPLICIT TYPE STATEMENT

The explicit type statement:

- Specifies the type and length of variables, arrays, and user-supplied functions.
- Specifies the dimensions of an array.

IBM EXTENSION

- Assigns initial data values for variables and arrays.

END OF IBM EXTENSION

The explicit type statement overrides the IMPLICIT statement, which, in turn, overrides the predefined convention for specifying type.

Syntax

```
type name1 [, name2 ] ...
```

type
: is COMPLEX, INTEGER, LOGICAL, REAL, DOUBLE PRECISION, or CHARACTER[*len[,]]

  where:

  *len
  : specifies the length (number of characters between 1 and 500). It is optional. It can be expressed as:

    - An unsigned, nonzero, integer constant.
    - An expression with a positive value that contains integer constants, names of integer constants enclosed in parentheses, or an asterisk enclosed in parentheses.

    The length *len immediately following the word CHARACTER is used as the length specification of any name in the statement that has no length specification attached to it. To override a length for a particular name, see the alternative forms of name below. If *len is not specified, it is assumed to be 1.

  The comma in CHARACTER[*len[,]] must not appear if *len is not specified. It is optional if *len is specified.

If the length specification (*len) is a constant, it must be an unsigned, nonzero, integer constant. If the length specification is an arithmetic expression enclosed in parentheses, it can contain only integer constants or names of integer constants. Function and array element references must not appear in the expression. The value of the expression must be a positive, nonzero, integer constant.

If the length specification is an asterisk (*), name must be the name of a character constant. The character constant assumes the length of its corresponding expression in a PARAMETER statement.

If the CHARACTER statement is in a subprogram, the asterisk (*) must be associated with a name that is a dummy argument. The dummy argument assumes the length of the associated actual argument for each reference of the subroutine.

The length specified (or assigned by default) with an array name is the length of each element of the array.

If a character function has the length specified as an asterisk (*) in a program unit, the function name must appear as the name of a function in a FUNCTION or ENTRY statement in the same program unit. When a reference to such a function is executed, the function assumes the length specified in the calling program unit. The length of a CHARACTER statement function cannot be specified by an asterisk (*) but can be an integer constant expression.

The length specified for a character function in the program unit that refers to the function must be an expression involving only integer constants or names of integer constants. This length must agree with the length specified in the subprogram that specifies the function if the length is not specified as an asterisk.

IBM EXTENSION

type
is COMPLEX[*len1], INTEGER[*len1], LOGICAL[*len1], or REAL[*len1]

where:

*len1
is optional and represents one of the permissible length specifications for its associated type as described in Figure 5.

END OF IBM EXTENSION

name
is a variable, array, function name, dummy procedure name or the name of a constant. It can have the form:

a[(dim)]
or
a[(dim)][*len2]

where:

a
is a variable, array, function name, or dummy procedure name.

dim
is optional. dim may only be specified for arrays. It is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array in the form:

e1:e2

or

e2

where:

e1
is the lower dimension bound. It is optional. If e1 (with its following colon) is not specified, its value is assumed to be 1.

e2
is the upper dimension bound and must always be specified.

(See "Size and Type Declaration of an Array" on page 22 for rules about dimension bounds.)

If a specific intrinsic function name appears in an explicit specification statement that causes a conflict with the type specified for this function in "Appendix B. FORTRAN-Supplied

Procedures" on page 204, the name loses its intrinsic function property in the program unit. A type statement that confirms the type of an intrinsic function is permitted. If a generic function name appears in an explicit specification statement, it does not lose its generic property in the program unit.

***len2**
overrides the length as specified in the statement by CHARACTER[*len[,]].

IBM EXTENSION

name
is a variable, array, function name, dummy procedure name or the name of a constant. It can have the form:

a[*len3][(dim)]

or

a[*len3][(dim)] [/i1,i2,i3,...,in/]

where:

a
is a variable, array, function name, or dummy procedure name.

***len3**
overrides the length as specified in the initial keyword of the statement as COMPLEX, INTEGER, LOGICAL, REAL, COMPLEX[*len1], INTEGER[*len1], LOGICAL[*len1], or REAL[*len1]

dim
is optional. dim may only be specified for arrays. It is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array. See the description of dim above.

i1,i2,i3,...,in
are optional and represent initial data values.

Dummy arguments and names of constants, functions, and statement functions, may not be assigned initial values. Initial data values may not be assigned for any items of type DOUBLE PRECISION or CHARACTER.

Initial data values may be assigned to variables or arrays that are not dummy arguments or in blank common, by use of in, where in is a constant or list of constants separated by commas. Each in provides initialization only for the immediately preceding variable or array. Lists of constants are used only to assign initial values to array elements. The data must be of the same type as the variable or array, except that hexadecimal data may also be used.

Note: If hexadecimal data is used, the hexadecimal constant form must be followed (see "Hexadecimal Constants" on page 17).

Successive occurrences of the same constant can be represented by the form i*constant, as in the DATA statement. If initial data values are assigned to an array in an explicit specification statement, the dimension information for the array must be in the explicit specification statement or in a preceding DIMENSION or COMMON statement.

END OF IBM EXTENSION

Valid Explicit Type Statements:

```
CHARACTER*8APPLES
DATA APPLES/'APPLES  '/
```

IBM EXTENSION

```
COMPLEX C,D/(2.1,4.7)/,E*16
INTEGER*2 ITEM/76/, VALUE
REAL A(5,5)/20*6.9E2,4*1.0/,B(100)/100*0.0/,TEST*8(5)/5*0.0D0/
REAL*8 BAKER, HOLD, VALUE*4, ITEM(5,5)
```

END OF IBM EXTENSION

EXTERNAL STATEMENT

The EXTERNAL statement identifies a user-supplied subprogram name and permits such a name to be used as an actual argument.

Syntax

```
EXTERNAL name1 [, name2 ] ...
```

name
is a name of a user-supplied subprogram (function or subroutine) that is passed as an argument to another subprogram.

EXTERNAL is a specification statement and must precede statement function definitions and all executable statements.

Statement function names cannot appear in an EXTERNAL statement. If the name of a VS FORTRAN-supplied function (that is, intrinsic function) is used in an EXTERNAL statement, the function is no longer recognized as being an intrinsic function when it appears as a function reference. Instead, it is assumed that the function is supplied by the user.

The same name may not appear in both an EXTERNAL and an INTRINSIC statement.

The name of any subprogram that is passed as an argument to another subprogram must appear in an EXTERNAL or INTRINSIC statement in the calling program.

Valid EXTERNAL Statement:

```
EXTERNAL TREES
```

FORMAT STATEMENT

The FORMAT statement is used with the input/output list in the READ and WRITE statements to specify the structure of FORTRAN records and the form of the data fields within the records.

Syntax

```
FORMAT (f1 [, f2 [ ... ] ] )
```

f1, f2,..., fn are format codes.

Format Codes	Description
aIw	Integer data fields
aIw.m	Integer data fields
aDw.d	Double precision data fields
aEw.d	Real data fields
aEw.dEe	Real data fields
aFw.d	Real data fields
aGw.d	Real data fields
aGw.dEe	Real data fields
nP	Scale factor
aLw	Logical data fields
aA	Character data fields
aAw	Character data fields
'literal'	Literal data (character constant)
wH	Literal data (Hollerith constant)
wX	Skip a field (input); fill with blanks (output)
Tr	Transfer of data starts in current position
TLr	Transfer of data starts r characters to the left of current position
TRr	Transfer of data starts r characters to the right of current position
a(...)	Group format specification
S	Display of optional plus sign is restored
SP	Plus sign is produced in output
SS	Plus sign is not produced in output
BN	Blanks are ignored in input
BZ	Blanks are treated as zeros in input
/	Data transfer on the current record is ended
:	Format control is terminated if there are no more items in the input/output list

IBM EXTENSION

Format Codes	Description
aEw.dDe	Real data fields
aGw.d	Integer or logical data fields
aGw.dEe	Integer or logical data fields
aQw.d	Extended precision data fields
aZw	Hexadecimal data fields

END OF IBM EXTENSION

a
is optional and is a repeat count, an unsigned, nonzero, integer constant used to denote the number of times the format code or group is to be used. If a is omitted, the code or group is used only once.

w
is an unsigned, nonzero, integer constant that specifies the width of the field.

m
is an unsigned integer constant that specifies the number of digits to be printed.

d
is an unsigned integer constant that specifies the number of digits to the right of the decimal point.

e
is an unsigned, nonzero, integer constant that specifies the number of digits in the exponent field.

n
is an (optionally) signed integer constant that specifies a scale factor to be applied.

r
is an unsigned, nonzero, integer constant that specifies a character position in a record.

(...)
is a group format specification. Within the parentheses are format codes or additional levels of groups, separated by commas, slashes, or colons. Commas are optional before or after a slash and before or after a colon, if the slash or colon is not part of a character constant.

The FORMAT statement is used with READ and WRITE statements for internal and external files. The external files must be connected for SEQUENTIAL or DIRECT access. In the FORMAT statement, the data fields are described with format codes, and the order in which these format codes are specified determines the structure of the FORTRAN records. The I/O list gives the names of the data items that make up the record. The length of the list, in conjunction

with the FORMAT statement, specifies the length of the record (see "Forms of a FORMAT Statement" on page 94).

The format codes delimited by left and right parentheses may appear as a character constant in the format specification of the READ or WRITE statement, instead of in a separate FORMAT statement. For example,

```
READ (UNIT=5,FMT='(I3,F5.2,E10.3,G10.3)')N,A,B,C

READ (5,'(I3,F5.2,E10.3,G10.3)')N,A,B,C
```

Throughout this section, the examples show punched card input and printed line output. However, the concepts apply to all input/output media. In the examples, the character b represents a blank.

General Rules for Data Conversion

The following is a list of general rules for using the FORMAT statement or a format in a READ or WRITE statement.

- FORMAT statements are not executed; their function is to supply information to the object program. They may be placed anywhere in a program unit other than in a block data subprogram, subject to the rules for the placement of the PROGRAM, FUNCTION, SUBROUTINE, and END statements.

- Complex data in records require two successive D, E, G, or F format codes.

  ┌──────── IBM EXTENSION ────────┐

  VS FORTRAN also accepts the Q format code for complex data.

  └──────── END OF IBM EXTENSION ────────┘

  The two codes may be different and the format codes T, TL, TR, X, /, :, S, SP, SS, P, BN, BZ, H, or a literal enclosed in apostrophes may appear between the two codes.

- When defining a FORTRAN record by a FORMAT, it is important to consider the maximum size record allowed on the input/output medium. For example, if a FORTRAN record is to be punched for output, the record should not be longer than 80 characters. If it is to be printed, it should not be longer than the printer's line length. For input, the FORMAT should not define a FORTRAN record longer than the actual input record.

- When formatted records are prepared for printing at a printer or terminal, the first character of the record is not printed or displayed. It is treated as a carrier control character. It can be specified in a FORMAT statement with either of two forms of literal data:

  'x' or 1Hx

  where x is one of the following:

x	Meaning
blank	Advance one line before printing.
0	Advance two lines before printing.
1	Advance to first line of next page.
+	Do not advance before printing. (Overstrike the current line.)

  For media other than a printer or terminal, the first character of the record is treated as data.

- If the I/O list is omitted from the READ or WRITE statement, the following general rules apply:

- **Input:** A record is skipped.
- **Output:** A blank record is written unless the FORMAT statement contains an H format code or a character constant (see "H Format Code and Character Constants" on page 103).

  To produce a blank record on output, an empty format specification of the form FORMAT ( ) may be used.

- To illustrate the nesting of group format specifications, the following statements are both valid:

```
FORMAT (...,a(...,a(...),...,a(...),...))
```

or

```
FORMAT (...,a(...,a(...,a(...),...),...),...)
```

- Names of constants must not be a part of a format specification (see "PARAMETER Statement" on page 138).
- With numeric data format codes I, F, E, G, and D, the following general rules apply:
  - **Input:** Leading blanks are not significant. The interpretation of blanks, other than leading blanks, is determined by a combination of the value of the BLANK= specifier given when the file was connected (see "OPEN Statement" on page 134) and any BN or BZ blank control that is currently in effect. Plus signs may be omitted. A field of all blanks is considered to be zero.

    With F, E, G, and D format codes, a decimal point appearing in the input field overrides the portion of a format code that specifies the decimal point location. The input field may have more digits than VS FORTRAN uses to approximate the value.
  - **Output:** The representation of a positive or zero internal value in the field may be prefixed with a plus, as controlled by the S, SP, and SS format codes. The representation of a negative internal value in the field is prefixed with a minus. A negative zero is not produced.

    The representation is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks are inserted in the field.

    If the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w*.*d*E*e* or G*w*.*a*E*e* format codes, the entire field of width *w* is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, asterisks are not produced. When an SP format code is in effect, a plus is not optional.

IBM EXTENSION

With VS FORTRAN, format code Q makes the following additional rules apply:

- **Input:** With Q editing, a decimal point appearing in the input field overrides the portion of a format code that specifies the decimal point location. The input field may have more digits than VS FORTRAN uses to approximate the value.
- **Output:** If the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w*.*d*D*e* or Q*w*.*d* format codes, the entire

field of width w is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, asterisks are not produced. When an SP format code is in effect, a plus is not optional.

END OF IBM EXTENSION

Forms of a FORMAT Statement

All of the format codes in a FORMAT statement are enclosed in parentheses. Within these parentheses, the format codes are delimited by commas. The comma may be omitted between a P format code and an immediately following F, E, D, or G format code, and before or after a colon or slash format code.

Execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information provided jointly by the I/O list, if one exists, and the format specification. If there is an I/O list, there must be at least one I, D, E, F, A, G, or L format code in the format specification.

IBM EXTENSION

The Q and Z format codes may also appear in the format specification.

END OF IBM EXTENSION

There is no I/O list item corresponding to the format codes: T, TL, TR, X, H, literals enclosed in apostrophes, S, SP, SS, BN, BZ, P, the slash (/), or the colon (:). These communicate information directly to the record.

Whenever an I, D, E, F, A, G, or L format code is encountered, format control determines whether there is a corresponding element in the I/O list.

IBM EXTENSION

With VS FORTRAN, the list of format codes includes Q and Z.

Whenever a Q or Z code is encountered, format control determines whether there is a corresponding element in the I/O list.

The comma may be omitted between a P format code and an immediately following Q format code.

END OF IBM EXTENSION

If there is a corresponding element, appropriately converted information is transmitted. If there is no corresponding element, the format control terminates, even if there is an unsatisfied repeat count.

When format control reaches the last (outer) right parenthesis of the format specification, a test is made to determine whether another element is specified in the I/O list. If not, control terminates. If another list element is specified, the format control starts a new record. Control then reverts to that group specification terminated by the last preceding right parenthesis, including its group repeat count, if any, or, if no group specification exists, then to the first left parenthesis of the format specification. Such a group specification must include a closing right parenthesis. If no group specification exists, control reverts to the first left parenthesis of the format specification.

For example, assume the following FORMAT statements:

```
70 FORMAT (I5,2(I3,F5.2),I4,F3.1)
```

```
80 FORMAT (I3,F5.2,2(I3,2F3.1))

90 FORMAT (I3,F5.2,2I4,5F3.1)
```

With additional elements in the I/O list after control has reached the last right parenthesis of each, control would revert to the 2(I3,F5.2) specification in the case of statement 70; to 2(I3,2F3.1) in the case of statement 80; and to the beginning of the format specification, I3,F5.2,... in the case of statement 90.

The question of whether there are further elements in the I/O list is asked only when an I, D, E, F, A, G, or L format code or the final right parenthesis of the format specification is encountered.

IBM EXTENSION

The question is also asked when a Q or Z format code is encountered.

END OF IBM EXTENSION

Before this is done, T, TL, TR, X, and H codes, literals enclosed in apostrophes, colons, and slashes are processed. If there are fewer elements in the I/O list than there are format codes, the remaining format codes are ignored.

I Format Code

The I format code **edits integer data**. For example, if a READ statement refers to a FORMAT statement containing I format codes, the input data is stored in internal storage in integer format. The magnitude of the data to be transmitted must not exceed the maximum magnitude of an integer constant.

INPUT: Leading blanks in a field of the input line are interpreted as zeros. Embedded and trailing blanks are treated as indicated in the general rules for numeric fields described under "General Rules for Data Conversion" on page 92. If the form I*w*.*m* is used, the value of *m* has no effect.

OUTPUT: If the number of significant digits and sign required to represent the quantity in the byte is less than *w*, the leftmost print positions are filled with blanks. If it is greater than *w*, asterisks are printed instead of the number. If the form I*w*.*m* is used, the output is the same as the I*w* form, except that the unsigned integer constant consists of at least *m* digits and, if necessary, has leading zeros. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

F Format Code

The F*w*.*d* format code **edits real data**. It indicates that the field occupies *w* positions, the fractional part of which consists of *d* digits.

INPUT: The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented.

The input field may have more digits than VS FORTRAN uses to approximate the value of the datum. The basic form may be followed by an exponent of one of the following forms:

- Signed integer constant.

- E followed by zero or more blanks, followed by an optionally signed integer constant.
- D followed by zero or more blanks, followed by an optionally signed integer constant.

IBM EXTENSION

- Q followed by zero or more blanks, followed by an optionally signed integer constant.

END OF IBM EXTENSION

An exponent containing a D is processed identically to an exponent containing an E.

OUTPUT: The output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, or an optional plus otherwise, followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to d fractional digits. Leading zeros are not provided except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero appears if there would otherwise be no digits in the output field.

D, E, and Q Format Codes

The Dw.d, Ew.d, Ew.dEe format codes **edit real, complex, or double precision data.**

IBM EXTENSION

The Ew.dDe and Qw.d format codes edit **extended precision data** in addition to real, complex, and double precision data.

END OF IBM EXTENSION

The external field occupies w positions, the fractional part of which consists of d digits (unless a scale factor greater than one is in effect). The exponent part consists of e digits. (The e has no effect on input.)

INPUT: The input field may have more digits than VS FORTRAN uses to approximate the value of the datum.

Input datum must be a number, which, optionally, may have a D or E exponent, or may be omitted from the exponent if the exponent is signed.

IBM EXTENSION

It may also have a Q exponent.

END OF IBM EXTENSION

All exponents must be preceded by a constant; that is, an optional sign followed by at least one decimal digit with or without decimal point. If the decimal point is present, its position overrides the position indicated by the d portion of the format code, and the number of positions specified by w must include a place for it. If the data has an exponent and a P format code is in effect, the scale factor is ignored.

The interpretation of blanks is explained in "General Rules for Data Conversion" on page 92.

The input datum may have an exponent of any form. The input datum is converted to the length of the variable as specified in the I/O list. The e of the exponent in the format code has no effect on input.

OUTPUT: For data written under a D or E format code, unless a P-scale factor is in effect, output consists of an optional sign (required for negative values), a decimal point, the number of significant digits specified by d, and a D or E exponent requiring four positions.

IBM EXTENSION

For data written under a Q format code, unless a P-scale factor is in effect, output consists of an optional sign (required for negative values), a decimal point, the number of significant digits specified by d, and a Q exponent requiring four positions.

END OF IBM EXTENSION

On output, w must provide sufficient space for an integer segment if it is other than zero, a fractional segment containing d digits, a decimal point, and, if the output value is negative, a sign. If insufficient space is provided for the integer portion, including the decimal point and sign (if any), asterisks are written instead of data. If excess space is provided, the number is preceded by blanks.

The fractional segment is rounded to d digits. A zero is placed to the left of the decimal point If the output field consists only of a fractional segment, and if additional space is available. If the entire value is zero, a zero is printed before the decimal point.

G Format Code

The G format code is a generalized code used to **transmit real data** according to the type specification of the corresponding variable in the I/O list.

INPUT: The form of the input field is the same as for the F format code.

OUTPUT: For real data, the d determines the number of digits to be printed and whether the number should be printed with the letter E or D followed by the exponent, depending on the length specification of the variable in the I/O list. The w specification for real data must include a position for a decimal point and, four positions for a decimal exponent, which includes the sign. A zero exponent has a plus sign. All other rules for output are the same as those for the individual format codes.

IBM EXTENSION

The letter Q is used for the exponent of real data.

The G format code may be used to transmit integer or logical data according to the type specification of the corresponding variable in the I/O list.

If the variable in the I/O list is integer or logical, the d portion of the format code, specifying the number of significant digits, can be omitted; if it is given, it is ignored.

END OF IBM EXTENSION

P Format Code

A P format code **specifies a scale factor** n, where n is an optionally signed integer constant. The value of the scale factor is zero at the beginning of execution of each input/output statement. It applies to all subsequently interpreted F, E, D, and G format codes until another scale factor is encountered, then that scale factor is established.

IBM EXTENSION

It also applies to all subsequently interpreted Q format codes.

END OF IBM EXTENSION

Reversion of format control does not affect the established scale factor. A repetition code can precede these format codes. For example, 2P3F7.4 is valid. A comma can be placed after the P format code, for example, 2P,3F7.4. A scale factor of zero may be specified.

INPUT: If an exponent is in the data field, the scale factor has no effect. If no exponent is in the field, the externally represented number equals the internally represented number multiplied by 10**n for the external representation.

For example, if the input data is in the form

```
  xx.xxxx
```

and is to be used internally in the form

```
    .xxxxxx
```

then the format code used to effect this change is

```
  2PF7.4
```

which may also be written 2P,F7.4.

Similarly, if the input data is in the form

```
    xx.xxxx
```

and is to be used internally in the form

```
  xxxx.xx
```

then the format code used to effect this change is

```
  -2PF7.4
```

which also may be written -2P,F7.4.

OUTPUT: With an F format code, the internally represented number reduced by 10**n is produced.

For example, if the number has the internal form

```
      .xxxxxx
```

and is to be written in the form

```
    xx.xxxx
```

the format code used to effect this change is

```
    2PF7.4
```

which also may be written 2P,F7.4.

On output with E and D format codes, the value of the internally represented number is not changed. When the decimal point is moved according to the d of the format code, the exponent is adjusted so that the value of the externally represented number is not multiplied by 10**n.

IBM EXTENSION

On output with Q format code, the value of the internally represented number is not changed.

END OF IBM EXTENSION

For example, if the internal number

238.47

were printed according to the format E10.3, it would appear as

0.238Eb03

If it were printed according to the format 1PE10.3 or 1P,E10.3 it would appear as

2.385Eb02

On output with a G format code, the effect of the scale factor is suspended unless the magnitude of the internally represented number (m) is outside the range that permits the use of F format code editing. This range for use of the F format code is

.1 > m > 10 ** d

where d is the number of digits as specified in the G format code Gw.d.

IBM EXTENSION

Z Format Code

The Z format code **transmits hexadecimal data.**

INPUT: Scanning of the input field proceeds from right to left. Leading, embedded, and trailing blanks in the field are treated as zeros. One byte in internal storage contains two hexadecimal digits; thus, if an input field contains an odd number of digits, the number is padded on the left with a hexadecimal zero when it is stored. If the storage area is too small for the input data, the data is truncated and high-order digits are lost.

OUTPUT: If the number of digits in the byte is less than w, the leftmost print positions are filled with blanks. If the number of digits in the byte is greater than w, the leftmost digits are truncated and the rest of the number is printed.

END OF IBM EXTENSION

Numeric Format Code Examples

Example 1:

The following example illustrates the use of format codes I, F, D, E, and G.

```
  75 FORMAT (I3,F5.2,E10.3,G10.3)

     READ (5,75) N,A,B,C
```

Explanation:

- Four input fields are described in the FORMAT statement and four variables are in the I/O list. Therefore, each time the READ statement is executed, one input line is read from the file connected to unit number 5.
- When an input line is read, the number in the first field of the line (three columns) is stored in integer format in location N. The number in the second field of the input line (five columns) is stored in real format in location A, and so on.
- If there were one more variable in the I/O list, say M, another line would be read and the information in the first three columns of that line would be stored in integer format in location M. The rest of the line would be ignored.
- If there were one fewer variable in the list (say C is omitted), format code G10.3 would be ignored.
- This FORMAT statement defines only one record format. "Forms of a FORMAT Statement" on page 94 explains how to define more than one record format in a FORMAT statement.

IBM EXTENSION

Example 2:

This example illustrates the use of the Z, D, and G format codes.

Assume that the following statements are given:

```
75 FORMAT (Z4,D10.3,2G10.3)

   READ (5,75) A,B,C,D
```

where A, C, and D are REAL*4 and B is REAL*8 and that on successive executions of the READ statement, the following input lines are read:

```
Column:    1   5          15         25         35

           v   v          v          v          v
           b3F1156432D+02276.38E+15bbbbbbbbbb
Input
           2AF3155381+02b382506E+28276.38E+15
Lines
           3ACb346.18D-03485.322836276.38E+15

Format:    Z4     D10.3      G10.3      G10.3
```

Then the variables A, B, C and D receive values as if the following data fields had been supplied:

A	B	C	D
03F1	156.432D02	276.38E+15	000000.000
2AF3	155.381+20	382.506E28	276.38E+15
3AC0	346.18D-03	485.322836	276.38E+15

Explanation:

- Leading blanks in an input field are treated as zeros. If all other blanks are assumed to be treated as zero, because the value for B on the second input line was not right justified in the field, the exponent is 20 not 2.
- Values read into the variables C and D with a G format code are converted according to the type of the corresponding variable in the I/O list.

END OF IBM EXTENSION

Example 3:

This example illustrates the use of the literal enclosed in apostrophes and the F, E, G, and I format codes.

Assume that the following statements are given:

```
 76 FORMAT   ('0',F6.2,E12.3,G14.6,I5)
    WRITE    (6,76)A,B,C,N
```

and that the variables A, B, C and N have the following values on successive executions of the WRITE statement:

A	B	C	N
034.40	123.380E+02	123.380E+02	031
031.1	1156.1E+02	123456789.	130
-354.32	834.621E-03	1234.56789	428
01.132	83.121E+06	123380.D+02	000

Then, the following lines are printed by successive executions of the WRITE statement:

```
Print
Column:    1         9              21              35
           v         v              v               v
            34.40    0.123E 05      12338.0          31
            31.10    0.116E 06      0.123457E 09     130
           ******    0.835E 00      1234.57          428
             1.13    0.831E 08      0.123380E 08     0
```

Explanation:

- The integer portion of the third value of A exceeds the format code specification, so asterisks are printed instead of a value. The fractional portion of the fourth value of A exceeds the format code specification, so the fractional portion is rounded.
- For the variable B the decimal point is printed to the left of the first significant digit and only three significant digits are printed because of the format code E12.3. Excess digits are rounded off from the right.
- The values of the variable C are printed according to the format specification G14.6. The s specification, which in this case is 6, determines the number of digits to be printed and whether the number should be printed with a decimal exponent. Values greater than or equal to 0.1 and less than 1000000 are printed without a decimal exponent in this example. Thus, the first and third values have no exponent.

The second and fourth values are greater than 1000000, so they are printed with an exponent.

L Format Code

The L format code **transmits logical variables.**

INPUT: The input field must consist of either zeros or blanks with an optional decimal point, followed by a T or F, followed by optional characters, for true and false, respectively. The T or F assigns a value of true or false to the logical variable in the input list. The logical constants .TRUE. and .FALSE. are acceptable input forms.

OUTPUT: A T or F is inserted in the output record depending upon whether the value of the logical variable in the I/O list was true or false, respectively. The single character is right justified in the output field and preceded by w-1 blanks.

A Format Code

The A format code **transmits character data.** Each alphabetic or special character is given a unique internal code. Numeric characters are transmitted without alteration; they are not converted into a form suitable for computation. Thus, the A format code can be used for numeric fields, but not for numeric fields requiring arithmetic.

If w is specified, the field consists of w characters.

If the number of characters w is not specified with the format code A, the number of characters in the field is the length of the character item in input/output list.

INPUT: The maximum number of characters stored in internal storage depends on the length of the variable in the I/O list. If w is greater than the variable length, say v, then the leftmost w-v characters in the field of the input line are skipped and remaining v characters are read and stored in the variable. If w is less than v, then w characters from the field in the input line are read and remaining rightmost characters in the variable are filled with blanks.

OUTPUT: If w is greater than the length v of the variable in the I/O list, then the printed field contains v characters right-justified in the field, preceded by leading blanks. If w is less than v, the leftmost w characters from the variable are printed and the rest of the data is truncated.

Example 1:

Assume that B has been specified as CHARACTER*8, that N and M are CHARACTER*4, and that the following statements are given:

```
25    FORMAT (3A7)

      READ   (5,25) B, N, M
```

When the READ statement is executed, one input line is read from the data set associated with data set reference number 5 into the variables B, N, and M in the format specified by FORMAT statement number 25. The following list shows the values stored for the given input lines (b represents a blank).

Input Line	B	N	M
ABCDEFG46bATb11234567	ABCDEFGb	ATb1	4567
HIJKLMN76543213334445	HIJKLMNb	4321	4445

Example 2:

Assume that A and B are character variables of length 4, that C is a character variable of length 8, and that the following statements are given:

```
26    FORMAT    (A6,A5,A6)

      WRITE     (6,26) A,B,C
```

When the WRITE statement is executed, one line is written on the data set associated with data set reference number 6 from the variables A, B, and C in the format specified by FORMAT statement 26. The printed output for values of A, B and C is as follows (b represents a blank):

A	B	C	Printed Line
A1B2	C3D4	E5F6G7H8	bbA1B2bC3D4E5F6G7

H Format Code and Character Constants

Character constants can appear in a FORMAT statement in one of two ways: following the H format code or enclosed in apostrophes. For example, the following FORMAT statements are equivalent.

```
25    FORMAT (22H 1981 INVENTORY REPORT)

25    FORMAT (' 1981 INVENTORY REPORT')
```

No item in the output list corresponds to the character constant. The constant is written directly from the FORMAT statement. (The FORMAT statement can contain other types of format code with corresponding variables in the I/O list.)

INPUT: Character constants cannot appear in a format used for input.

OUTPUT: The character constant from the FORMAT statement is written on the output file. (If the H format code is used, the w characters following the H are written. If apostrophes are used, the characters enclosed in apostrophes are written.) For example, the following statements:

```
8 FORMAT (14H0MEAN AVERAGE:, F8.4)

  WRITE (6,8) AVRGE
```

would write the following record if the value of AVRGE were 12.3456:

```
MEAN AVERAGE: 12.3456
```

The first character of the output data record in this example is the carrier control character for printed output. One line is skipped before printing, and the carrier control character does not appear in the printed line.

Note: If the character constant is enclosed in apostrophes, an apostrophe character in the data is represented by two successive apostrophes. For example, DON'T would be represented as 'DON''T'.

X Format Code

The X format code **specifies a field of w characters to be skipped** on input or filled with blanks on output if the field was not previously filled. On output, an X format code does not affect the length of a record. For example, the following statements:

- Read the first ten characters of the input line into variable I.

- Skip over the next ten characters without transmission.
- Read the next four fields of ten characters each into the variables J, K, L, and M.

```
5    FORMAT (I10,10X,4I10)

     READ   (5,5) I,J,K,L,M
```

T Format Code

The T format code **specifies the position in the FORTRAN record at which the transfer of data is to begin.**

To illustrate the use of the T code, the following statements:

```
5    FORMAT (T40,'1981 STATISTICAL REPORT', T80,
    X  'DECEMBER',T1,'0PART NO. 10095')
      WRITE  (6,5)
```

print the following:

```
Print
Position:  1                    39                          79

           v                    v                           v
           PART NO. 10095       1981 STATISTICAL REPORT     DECEMBER
```

The T format code can be used in a FORMAT statement with any type of format code, as, for example, with FORMAT ('0',T40,I5).

INPUT: The T format code allows portions of a record to be processed more than once, possibly with different format codes.

OUTPUT: The record is assumed to be initially filled with blank characters, and the T format code can replace or skip characters. On output, a T format code does not affect the length of a record.

(For printed output, the first character of the output data record is a carrier control character and is not printed. Thus, for example, if T50,'Z' is specified in a FORMAT statement, a Z will be the 50th character of the output record, but it will appear in the 49th print position.)

TL AND TR FORMAT CODES: The TL and TR format codes specify how many characters left (TL) or right (TR) from the current character position the transfer of data is to begin. With TL format code, if the current position is less than or equal to the position specified with TL, the next character transmitted will be placed in position 1 (that is, the carrier control position).

The TL and TR format codes can be used in a FORMAT statement with any type of format code. On output, these format codes do not affect the length of a record.

Group Format Specification

The group format specification **repeats a set of format codes and controls the order in which the format codes are used.**

The group repeat count a is the same as the repeat indicator a that can be placed in front of other format codes. For example, the following statements are equivalent:

```
10   FORMAT  (I3,2(I4,I5),I6)

10   FORMAT  (I3,(I4,I5,I4,I5),I6)
```

Group repeat specifications control the order in which format codes are used, since control returns to the last group repeat specification when there are more items in the I/O list than there are format codes in the FORMAT statement (see "Forms of a FORMAT Statement" on page 94). Thus in the previous example, if there were more than six items in the I/O list, control would return to the group repeat count 2 that precedes the specification (I4,I5).

If the group repeat count is omitted, a count of 1 is assumed. For example, the statements:

```
15    FORMAT  (I3,(F6.2,D10.3))

      READ    (5,15) N,A,B,C,D,E
```

read values from the first record for N, A, and B, according to the format codes I3,F6.2, and D10.3, respectively. Then, because the I/O list is not exhausted, control returns to the last group repeat specification, the next record is read, and values are transmitted to C and D according to the format codes F6.2 and D10.3, respectively. Since the I/O list is still not exhausted, another record is read and value is transmitted to E according to the format code F6.2—the format code D10.3 is not used.

All format codes can appear within the group repeat specification. For example, the following statement is valid:

```
40    FORMAT  (2I3/(3F6.2,F6.3/D10.3,3D10.2))
```

The first physical record, containing two data items, is transmitted according to the specification 2I3; the second, fourth, and so on, records, each containing four data items, are transmitted according to the specification 3F6.2,F6.3; and the third, fifth, and so on, records, each also containing four data items, are transmitted according to the specification D10.3,3D10.2, until the I/O list is exhausted.

S, SP, and SS Format Codes

The S, SP, and SS format codes **control optional plus characters in numeric output fields**. At the beginning of execution of each formatted output statement, a plus is produced in numeric output fields. If an SP format code is encountered in a format specification, a plus is produced in any subsequent position that normally contains an optional plus. If SS is encountered, a plus is not produced in any subsequent position that normally contains an optional plus. If an S is encountered, the option of producing the plus is restored.

The S, SP, and SS format codes affect only I, F, E, G, and D editing during the execution of an output statement.

IBM EXTENSION

The S, SP, and SS format codes also affect Q editing.

END OF IBM EXTENSION

The S, SP, and SS format codes have no effect during the execution of an input statement.

BN Format Code

The BN format code **specifies the interpretation of blanks**, other than leading blanks, in numeric input fields. At the beginning of each formatted input statement, such blank characters are interpreted as zeros or are ignored depending on the value of the BLANK= specifier given when the unit was connected (see "OPEN Statement" on page 134).

If BN is encountered in a format specification, all such blank characters in succeeding numeric input fields are ignored. However, a field of all blanks has the value zero.

The BN format code affects only I, F, E, G, and D editing during execution of an input statement.

IBM EXTENSION

The BN format code also affects Q editing during execution of an input statement.

END OF IBM EXTENSION

The BN format code has no effect during execution of an output statement.

BZ Format Code

The BZ format code **specifies the interpretation of blanks**, other than leading blanks, in numeric input fields.

If BZ is encountered in a format specification, all nonleading blank characters in succeeding numeric fields are treated as zeros. If no OPEN statement is given and the file is preconnected, all nonleading blanks in numeric fields are interpreted as zeros.

The BZ format code affects only I, F, E, G, and D editing during execution of an input statement.

IBM EXTENSION

The BZ format code also affects Q editing during execution of an input statement.

END OF IBM EXTENSION

The BZ format code has no effect during execution of an output statement.

Slash Format Code

A slash **indicates the end of a FORTRAN record.**

On input from a file connected for sequential access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record.

On output to a file connected for sequential access, a new record is created. For example, on output, the statement:

```
25   FORMAT   (I3,F6.2/D10.3,F6.2)
```

describes two FORTRAN record formats. The first, third, etc., records are transmitted according to the format I3, F6.2 and the second, fourth, etc., records are transmitted according to the format D10.3, F6.2.

Consecutive slashes can be used to introduce blank output records or to skip input records. If there are n consecutive slashes at the beginning or end of a FORMAT statement, n input records are skipped or n blank records are inserted between output records. If n consecutive slashes appear anywhere else in a FORMAT statement, the number of records skipped or blank records inserted is n-1. For example, the statement:

```
25   FORMAT   (1X,10I5//1X,8E14.5)
```

describes three FORTRAN record formats. On output, it places a blank line between the line written with format 1X,10I5 and the line written with the format 1X,8E14.5.

For a file connected for direct access, when a slash is encountered, the record number is increased by one and the file is positioned at the beginning of the record that has that record number.

Colon Format Code

A colon **terminates format control** if there are no more items in the input/output list. The colon has no effect if there are more items in the input/output list.

Example:

Assume the following statements:

```
       ITABLE=10
       IELEM=0
         .
         .
         .
    10 WRITE(6,1000)ITABLE,IELEM
         .
         .
         .
       ITABLE=11
       IELEM=25
         .
         .
         .
       XMIN=.37E1
       XMAX=.2495E3
         .
         .
         .
    20 WRITE(6,1000)ITABLE,IELEM,XMIN,XMAX
  1000 FORMAT('0 TABLE NUMBER',I5,:,'CONTAINS',I5,'ELEMENTS',:,
     1      /'MINIMUM VALUE:',E15.7,
     2      /'MAXIMUM VALUE:',E15.7)
```

The WRITE statement at statement number 10 generates the following:

```
 TABLE NUMBER 10 CONTAINS 0 ELEMENTS
```

The WRITE statement at statement number 20 generates the following:

```
 TABLE NUMBER 11 CONTAINS 25 ELEMENTS
 MINIMUM VALUE: -.3700000E+01
 MAXIMUM VALUE:  .2495000E+03
```

Reading Format Specifications at Object Time

FORTRAN provides for variable FORMAT statements by allowing a format specification to be read into a character array element or a character variable in storage. The data in the character array or variable may then be used as the format specification for subsequent input/output operations. The format specification may also be placed into the character array or variable by a DATA statement or an explicit specification statement in the source program. The following rules are applicable:

- The format specification must be a character array or character variable, even if the array size is only 1.
- The format codes entered into the array or character variable must have the same form as a source program FORMAT statement, except that the word FORMAT and the statement number are omitted. The parentheses surrounding the format codes are required.

- If a format code read at object time contains two consecutive apostrophes within a character field that is defined by apostrophes, it should be used for output only.
- Blank characters may precede the format specification, and character data may follow the right parenthesis that ends the format specification.

Example: Assume the following statements:

```
      DIMENSION C(5)
      CHARACTER*16 FMT
      READ(5,1)FMT
    1 FORMAT (A)
      READ(5,FMT)A,B,(C(I),I=1,5)
```

Assume, also, that the first input line assoicated with unit 5 contains (2E10.3, 5F10.8).

The data on the next input line is read, converted, and stored in A,B, and the array C, according to the format codes 2E10.3, 5F10.8.

IBM EXTENSION

READING A FORMAT INTO A NONCHARACTER ARRAY

Assume the following statements:

```
      DIMENSION FMT(16),C(5)
      READ(5,1) FMT
    1 FORMAT(16A1)
      READ(5,FMT)A,B,(C(I),I=1,5)
```

Assume also that the first input line associated with unit 5 contains (2E10.3, 5F10.8).

The data on the next input record is read, converted, and stored in A, B, and the array C, according to the format codes 2E10.3, 5F10.8.

END OF IBM EXTENSION

List-Directed Formatting

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a constant, a null value, or one of the forms:

*r***f*

or

*r**

where *r* is an unsigned, nonzero, integer constant. The *r***f* form is equivalent to *r* successive appearances of the constant *f*, and the *r** form is equivalent to *r* successive null values. Neither of these forms may contain embedded blanks except where permitted within the constant *f*.

A value separator is one of the following:

- A comma, optionally preceded by one or more blanks and optionally followed by one or more blanks

- A slash, optionally preceded by one or more blanks and optionally followed by one or more blanks
- One or more blanks between two constants or following the last constant

INPUT: Input forms acceptable to format specifications for a given type are acceptable for list-directed formatting, except as noted below. The form of the input value must be acceptable for the type of the input list item. Blanks are never treated as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below. The end of a record has the effect of a blank, except when it appears within a character constant.

When the corresponding input list item is of type real or double precision, the input form is that of a numeric input field. A numeric input field is a field suitable for the F format code that is assumed to have no fractional digits unless a decimal point appears within the field.

When the corresponding list item is of type complex, the input form consists of a left parenthesis, an ordered pair of numeric input fields separated by a comma, and a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of type logical, the input form must not include either slashes or commas among the optional characters permitted for the L format code.

When the corresponding list item is of type character, the input form consists of a nonempty string of characters enclosed in apostrophes. Each apostrophe within a character constant must be represented by two consecutive apostrophes without an intervening blank or end of record. Character constants may be continued from the end of one record to the beginning of the next record. The end of the record does not cause a blank or any other character to become part of the constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

For example, let *len* be the length of the list item, and let *w* be the length of the character constant. If *len* is less than or equal to *w*, the leftmost *len* characters of the constant are transmitted to the list item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the list item and the remaining *len*-*w* characters of the list item are filled with blanks. The effect is that the constant is assigned to the list item in a character assignment statement.

A null value is specified by having no characters between successive separators, by having no characters preceding the first value separator in the first record read by each execution of a list-directed input statement, or the *r** form. A null value has no effect on the definition status by the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value may not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant. The end of a record following any other separator, with or without separating blanks, does not specify a null value.

A slash encountered as a value separator during execution of a list-directed input statement causes termination of execution of that input statement after the assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them.

All blanks in a list-directed input record are considered part of some value separator, except for the following:

- Blanks embedded in a character constant
- Embedded blanks surrounding the real or imaginary part of a complex constant
- Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma

OUTPUT: The form of the values produced is the same as that required for input, except as noted. With the exception of character constants, the values are separated by one of the following:

- One or more blanks
- A comma, optionally preceded by one or more blanks and optionally followed by one or more blanks

VS FORTRAN may begin new records as necessary but, except for complex constants and character constants, the end of a record must not occur within a constant, and blanks must not appear within a constant.

Logical output constants are T for the value .TRUE. and F for the value .FALSE..

Integer output constants are produced with the effect of an I*w* edit descriptor for some reasonable value of *w*.

Real and double precision constants are produced with the effect of either an F format code or an E format code, depending on the magnitude *x* of the value and a range:

$$10^{**}d1 \leq 10^{**}d2$$

where *d1* and *d2* are processor-dependent integer values. If the magnitude *x* is within this range, the constant is produced using 0PF*w*.*d*; otherwise, 1PE*w*.*d*E*e* is used. Reasonable processor-dependent values are used for each of the cases involved.

Complex constants are enclosed in parentheses, with a comma separating the real and imaginary parts. The end of a record may occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.

Character constants produced:

- Are not delimited by apostrophes
- Are not preceded or followed by a value separator
- Have each internal apostrophe represented externally by one apostrophe
- Have a blank character inserted at the beginning of any record that begins with the continuation of a character constant from the preceding record

If two or more successive values in an output record produced have identical values, the sequence of identical values are written.

Slashes, as value separators, and null values are not produced by list-directed formatting.

Each output record begins with a blank character to provide carrier control if the record is printed.

UNCTION STATEMENT

The FUNCTION statement identifies a function subprogram. A function subprogram consists of a FUNCTION statement followed by other statements including at least one RETURN statement. It is an independently written program that is executed wherever its name is referred to in another program.

Syntax

```
[type] FUNCTION name ([arg1 [, arg2] ... ] )
```

type
is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER[*len1]

where:

***len1**
is the length specification. It is optional; if omitted, it is assumed to be 1. It may be an unsigned, nonzero, integer constant, an integer constant expression enclosed in parentheses, or an asterisk enclosed in parentheses. The expression can only contain integer constants; it must not include names of integer constants.

If the name is of type CHARACTER, all entry names must be of type CHARACTER, and lengths must be the same. If one length is specified as an asterisk, all lengths must be specified as an asterisk.

name
is the name of the function.

IBM EXTENSION

name*len2
is the name of the function.

where:

***len2**
is a positive, nonzero, unsigned integer constant. It represents one of the permissible length specifications for its associated type. (See "Variable Types and Lengths" on page 18.) It may be included optionally only when type is specified. It must not be used when DOUBLE PRECISION or CHARACTER type is specified.

END OF IBM EXTENSION

arg
is a dummy argument. It must be a variable or array name that may appear only once within the FUNCTION statement or dummy procedure name. If there is no argument, the parentheses must be present. (See "Dummy Arguments in a Function Subprogram" on page 113.)

A type declaration for a function name may be made by the predefined convention, by an IMPLICIT statement, by an explicit specification in the FUNCTION statement, or by an explicit type specification statement within the function subprogram. If the type of a function is specified in a FUNCTION statement, the function name must not appear in an explicit type specification statement.

The name of a function must not be in any other nonexecutable statement except a type statement.

Because the FUNCTION statement is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

The FUNCTION statement must be the first statement in the subprogram. The function subprogram may contain any FORTRAN statement except a SUBROUTINE statement, another FUNCTION statement, a BLOCK DATA statement, or a PROGRAM statement. If an IMPLICIT statement is used in a function subprogram, it must follow the FUNCTION statement and may only be preceded by another IMPLICIT statement, a PARAMETER, FORMAT, or ENTRY statement.

The name of the function (or one of the ENTRY names) must appear as a variable name in the function subprogram and must be assigned a value at least once during the execution of the subprogram in one of the following ways:

- As the variable name to the left of the equal sign in an arithmetic, logical, or character assignment statement
- As an argument of a CALL statement that will cause a value to be assigned in the subroutine referred to
- In the list of a READ statement within the subprogram
- As one of the parameters in an INQUIRE statement that is assigned a value within the subprogram

The value of the function is the last value assigned to the name of the function when a RETURN or END statement is executed in the subprogram. For additional information on RETURN and END statements in a function subprogram, see "RETURN Statement" on page 164 and "END Statement" on page 77.

The function subprogram may also use one or more of its arguments to return values to the calling program. An argument so used must appear:

- On the left side of an arithmetic, logical, or character assignment statement
- In the list of a READ statement within the subprogram
- As an argument in a function reference that is assigned a value by the function referred to
- As an argument in a CALL statement that is assigned a value in the subroutine referred to
- As one of the parameters in an INQUIRE statement

The dummy arguments of the function subprogram (for example, arg1, arg2, arg3,..., argn) are replaced at the time of invocation by the actual arguments supplied in the function reference in the calling program.

If a function dummy argument is used as an adjustable array name, the array name and all the variables in the array declarators (except those in COMMON) must be in the dummy argument list. See "Size and Type Declaration of an Array" on page 22.

If the predefined convention is not correct, the function name must be typed in the program units that refer to it. The type and length specifications of the function name in the function reference must be the same as those of the function name in the FUNCTION statement.

Except in a character assignment statement, the name of a character function whose length specification is an asterisk must not be the operand of a concatenation operation.

The length specified for a character function in the program unit that refers to the function must agree with the length specified in the subprogram that specifies the function. There is always agreement of length if the asterisk is used in the referenced subprogram to specify the length of the function.

Actual Arguments in a Function Subprogram

The actual arguments in a function reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced function. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type.

If an actual argument is of type character, the associated dummy argument must be of type character and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument is greater than the length of an associated dummy argument, the leftmost characters of the actual argument are associated with the dummy argument.

An actual argument in a function reference must be one of the following:

- An array name
- An intrinsic function name
- An external procedure name
- A dummy argument name
- An expression, except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is the name of a constant.

For an entry point in a function subprogram, see "ENTRY Statement" on page 81.

Dummy Arguments in a Function Subprogram

The dummy arguments of a function subprogram appear after the function name and are enclosed in parentheses. They are replaced at the time of invocation by the actual arguments supplied in the function reference.

Dummy arguments must adhere to the following rules:

- None of the dummy argument names may appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, SAVE, INTRINSIC, or NAMELIST statement, except as NAMELIST or common block names, in which case the names are not associated with the dummy argument names.
- A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, ENTRY or statement function definition in the same program unit.
- The dummy arguments must correspond in number, order, and type to the actual arguments.
- If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, an array element, a substring, or an array. A constant, name of constant, subprogram name, or expression should not be written as an actual argument unless the programmer is certain that the corresponding dummy argument is not assigned a value in the subprogram.

- A referenced subprogram cannot assign new values to dummy arguments that are associated with other dummy arguments within the subprogram or with variables in COMMON.

GO TO STATEMENTS

GO TO statements transfer control to an executable statement in the program unit. There are three GO TO statements:

- Assigned GO TO statement
- Computed GO TO statement
- Unconditional GO TO statement

Assigned GO TO Statement

The assigned GO TO statement transfers control to the statement numbered stn1, stn2, stn3 ..., depending on whether the current assignment of i is stn1, stn2, stn3 ..., respectively. (See "ASSIGN Statement" on page 46.)

Syntax

```
GO TO i [ [,] (stn1 [,stn2] [,stn3] ... ) ]
```

i
: is an integer variable (not an array element) of length 4 that has been assigned a statement number by an ASSIGN statement.

stn
: is the number of an executable statement in the program unit containing the assigned GO TO statement.

The list of statement numbers, that is, (stn1, stn2 ...), is optional. If omitted, the preceding comma must be omitted. If the list of statement numbers is specified, the preceding comma is optional. The statement number assigned to i must be one of the statement numbers in the list. The statement number may appear more than once in the list.

The ASSIGN statement that assigns the statement number to i must appear in the same program unit as the assigned GO TO statement that is using this statement number.

For example, in the statement:

```
GO TO N, (10, 25, 8)
```

If the current assignment of the integer variable N is statement number 8, then the statement numbered 8 is executed next. If the current assignment of N is statement number 10, the statement numbered 10 is executed next. If N is assigned statement number 25, statement 25 is executed next.

At the time of execution of an assigned GO TO statement, the current value of i must have been assigned the statement number of an executable statement (not a FORMAT statement) by the previous execution of an ASSIGN statement.

If at the time of the execution of an assigned GO TO statement, the current value of i contains an integer value, assigned directly or through EQUIVALENCE, COMMON, or argument passing, the result of the GO TO is unpredictable. Also, the integer variable i may not be a dummy argument in a subprogram. An integer variable may not be used as an actual argument in a subprogram reference at the time it is assigned a number.

Any executable statement immediately following the assigned GO TO statement should have a statement number; otherwise, it can never be referred to or executed.

Example:

```
ASSIGN 150 TO IASIGN
IVAR=150.
GO TO IASIGN
```

Computed GO TO Statement

The computed GO TO statement transfers control to the statement numbered stn1, stn2, or stn3,... depending on whether the current value of m is 1, 2, or 3,... respectively.

Syntax

```
GO TO (stn1 [, stn2] [, stn3] ... ) [,] m
```

stn
is the number of an executable statement in the program unit containing the computed GO TO statement. The same number may appear more than once within the parentheses.

m
is an integer expression. The comma before m is optional. If the value of m is outside the range 1 ≤ m ≤ n, the next statement is executed.

Example:

```
171  GO TO(172,173,174,173) INT(A)
172  A = A + 1.0
     GO TO 174
173  A = A + 1.0
174  CONTINUE
```

Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the statement specified by the statement number. Every subsequent execution of this GO TO statement results in a transfer to that same statement.

Syntax

```
GO TO stn
```

stn
is the number of an executable statement in the program unit containing the unconditional GO TO statement.

Any executable statement immediately following this statement must have a statement number; otherwise, it can never be referred to or executed.

Example:

```
      GO TO 5
999   I = I + 200
        .
        .
        .
  5   I = I + 1
```

IF STATEMENTS

The IF statements specify alternative paths of execution depending on the condition given. There are three forms of the IF statement:

- Arithmetic IF
- Block IF

  END IF
  ELSE
  ELSE IF

- Logical IF

Arithmetic IF Statement

The arithmetic IF statement transfers control to the statement numbered stn1, stn2, or stn3 when the value of the arithmetic expression (m) is less than, equal to, or greater than zero, respectively. The same statement number may appear more than once within the same IF statement.

Syntax

```
IF (m) stn1, stn2, stn3
```

m
is an arithmetic expression of any type except complex.

stn
is the number of an executable statement in the program unit containing the IF statement.

Any executable statement immediately following this statement must have a statement number; otherwise, it can never be referred to or executed.

Block IF Statement

The block IF statement is used with the END IF statement and, optionally, the ELSE IF and ELSE statements to control the execution sequence.

Syntax

```
IF (m) THEN
```

m
is any logical expression.

Two terms are used in connection with the block IF statement, **IF-level** and **IF-block**.

IF-level The number of IF-levels in a program unit is determined by the number of sets of block-IF statements (IF (m) THEN and END IF statements).

The IF-level of a particular statement (stn) is determined with the formula:

n1 - n2.

where:

n1
is the number of block IF statements from the beginning of the program unit up to and including the statement (*stn*).

n2
is the number of END IF statements in the program unit up to, but not including, the statement (*stn*).

IF-block An *IF-block* begins with the first statement after the block IF statement (IF (*m*) THEN), ends with the statement preceding the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement, and includes all the executable statements in between. An IF-block is empty if there are no executable statements in it.

Transfer of control into an IF-block from outside the IF-block is prohibited.

Execution of a block IF statement evaluates the expression *m*. If the value of *m* is true, normal execution sequence continues with the first statement of the IF-block, which is immediately following the IF (*m*) THEN. If the value of *m* is true, and the IF-block is empty, control is transferred to the next END IF statement that has the same IF-level as the block IF statement. If the value of *m* is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement.

If the execution of the last statement in the IF-block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the block IF statement that precedes the IF-block.

A block IF statement cannot terminate the range of a DO.

END IF Statement

The END IF statement concludes an IF-block. Normal execution sequence continues.

Syntax

```
END IF
```

For each block IF statement, there must be a matching END IF statement in the same program unit. A matching END IF statement is the next END IF statement that has the same IF-level as the block IF statement.

An END IF statement cannot terminate the range of a DO. Execution of an END IF statement has no effect.

Example:

```
IF (m) THEN
   .
   .
   .
END IF
```

ELSE Statement

The ELSE statement is executed if the preceding block IF or ELSE IF condition is evaluated as FALSE. Normal execution sequence continues.

Syntax

```
ELSE
```

An ELSE-block consists of all the executable statements after the ELSE statement up to, but not including, the next END IF statement that has the same IF-level as the ELSE statement. An ELSE-block may be empty.

Within an IF block, you can have only one ELSE.

Transfer of control into an ELSE-block from outside the ELSE-block is prohibited. The statement number, if any, of an ELSE statement must not be referred to by any statement (except an AT statement of a DEBUG packet). An ELSE statement cannot terminate the range of a DO.

Example:

```
IF (m) THEN
   .
   .
   .
ELSE
   .
   .
   .
END IF
```

ELSE IF Statement

The ELSE IF statement is executed if the preceding block IF condition is evaluated as false.

Syntax

```
ELSE IF (m) THEN
```

m
is any logical expression.

An ELSE IF-block consists of all of the executable statements after the ELSE IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement. An ELSE IF-block may be empty.

If the value of the logical expression *m* is true, normal execution sequence continues with the first statement of the ELSE IF-block.

If the value of *m* is true and the ELSE IF-block is empty, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement.

If the value of *m* is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement.

Transfer of control into an ELSE IF-block from outside the ELSE IF-block is prohibited. The statement number (*stn*), if any, of the

ELSE IF statement must not be referred to by any statement (except an AT statement of a DEBUG packet).

If execution of the last statement in the ELSE IF-block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement that precedes the ELSE IF-block.

An END IF statement cannot terminate the range of a DO.

Example 1:

```
IF (m) THEN
   .
   .
   .
ELSE IF (m) THEN
   .
   .
   .
END IF
```

Example 2:

```
IF (m) THEN
   .
   .
   .
ELSE IF (m) THEN
   .
   .
   .
ELSE
END IF
```

Logical IF Statement

The logical IF statement evaluates a logical expression and executes or skips a statement, depending on whether the value of the expression is true or false, respectively.

Syntax

```
IF (m) st
```

m
is any logical expression.

st
is any executable statement except a DO statement, another logical IF statement, an END statement, a block IF, ELSE IF, ELSE, or END IF statement.

IBM EXTENSION

st may not be a TRACE ON, TRACE OFF, INCLUDE, or DISPLAY statement.

END OF IBM EXTENSION

The statement st must not have a statement number.

The execution of a function reference in m is permitted to affect entities in the statement st.

The logical IF statement containing st may have a statement number.

Examples:

```
      IF(A.LE.0.0) GO TO 25
      C = D + E
      IF (A.EQ.B) ANSWER = 2.0*A/C
      F = G/H
   25 W = X**Z
         .
         .
         .
```

IMPLICIT TYPE STATEMENT

The IMPLICIT type statement specifies the type and length of all variables, arrays, and user-supplied functions whose names begin with a particular letter. It may be used to change or confirm implicit typing.

Syntax

```
IMPLICIT type (a [, a ]...) [, type (a [, a]...) ] ...
```

type
is CHARACTER[*len1], COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, or REAL

where:

***len1**
can be an unsigned, nonzero, integer constant or a positive integer constant expression enclosed in parentheses. It is optional.

If len1 is not specified, the length is one.

IBM EXTENSION

type
is COMPLEX[*len2], INTEGER[*len2], LOGICAL[*len2], or REAL[*len2]

where:

***len2**
can be a positive, nonzero, unsigned, integer constant. It represents one of the permissible length specifications for its associated type. It is optional.

END OF IBM EXTENSION

a
is a single alphabetic character or a range of characters drawn from the set A, B,..., Z. The range is denoted by the first and last characters of the range separated by a minus sign (for example, A-D).

IBM EXTENSION

The alphabetic character a can also be the currency symbol ($). The currency symbol ($) follows the letter Z. Thus, the range Y-$ is the same as Y,Z,$.

END OF IBM EXTENSION

The IMPLICIT specification statement can only be preceded by a PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA, PARAMETER, ENTRY, or FORMAT statement, or another IMPLICIT statement. The IMPLICIT specification statement declares the type of the variables and user-supplied functions appearing in this program (that is, integer, real, complex, logical, or character) by specifying that names beginning with certain designated letters are of a certain type. Furthermore, the IMPLICIT statement allows the programmer to declare the number of bytes to be allocated for each in the group of specified variables.

When type is CHARACTER, the length specification is between 1 and 500. The standard (default) length is 1.

The type and length associated with a letter or a range of letters must not conflict with the type or length given previously to the

same letters in the same IMPLICIT statement, in a different IMPLICIT statement or in a PARAMETER statement. Type specification by an IMPLICIT statement may be overridden or confirmed for any particular variable, array, name of a constant, external function, or statement function name by the appearance of that name in an explicit type specification statement.

(See "Type Declaration by the Predefined Specification" on page 20.)

Note: An IMPLICIT statement has no effect on names of FORTRAN-supplied (intrinsic) functions.

Valid IMPLICIT Statements:

```
IMPLICIT INTEGER(A-H), REAL(I-K), LOGICAL(L,M,N)

IMPLICIT COMPLEX(C-F)
```

IBM EXTENSION

```
IMPLICIT INTEGER(W-$)
```

All names beginning with W, X, Y, Z, and $ are considered integers of length 4 bytes.

END OF IBM EXTENSION

IBM EXTENSION

INCLUDE STATEMENT

The INCLUDE statement is a compiler directive. It inserts a specified statement or a group of statements into a program unit.

Syntax

```
INCLUDE (name)
```

name
is the name of a group of one or more FORTRAN source statements to be inserted into the source program being compiled.

The group must reside in a library known to the VS FORTRAN compiler.

The following rules apply to the INCLUDE statement:

- INCLUDE is a compile-time control statement only.
- The INCLUDE statement may not be continued.
- No replacement or editing is done.
- The inserted group may contain any complete VS FORTRAN source statement, including another INCLUDE statement.
- An INCLUDE of a group may not contain an INCLUDE statement that refers to a currently open INCLUDE group (that is, recursion is not permitted).
- Multiple INCLUDE statements may appear in the original source program.
- INCLUDE statements may appear anywhere in a source program before the END statement, except as the trailer of a logical IF statement. An END statement may be part of the included group.
- The FORTRAN statements in the group being included must be in the same form as the source program being compiled; that is, fixed form or free form.
- The resulting FORTRAN program after the inclusion of all groups must follow all FORTRAN rules as to sequence of statements.

END OF IBM EXTENSION

INQUIRE STATEMENT

An INQUIRE statement supplies information about properties of a particular named external file or of the connection to a particular external unit.

There are two forms of the INQUIRE statement:

- Inquire by **file name**
- Inquire by **unit number**

A sequential file or a direct-access file can be queried about its existence, its connection to a unit, its unit number, its name, its access method, whether it is formatted or unformatted, and how blanks are to be interpreted. In addition, a direct-access file is queried about its record length or its next record number.

The INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values assigned by the INQUIRE statement are those that are current at the time the statement is executed. All value assignments are done according to the rules for assignment statements. No error is given if the value is truncated because the receiving field is too small to contain it all.

INQUIRE by File Name

This INQUIRE statement supplies information about a file. When this statement is executed, the file specified by fn may or may not be connected to a unit. If the file is connected to a unit, the file may or may not exist. (For example, an output unit may be connected to a file but no output has been written.)

Syntax

```
INQUIRE (FILE=fn [, ERR=stn] [, IOSTAT=ios] [, EXIST=exs]
                 [, OPENED=opn] [, NAMED=nmd] [, NAME=nam]
                 [, SEQUENTIAL=seq] [, DIRECT=dir]
                 [, FORMATTED=fmt] [, UNFORMATTED=unf]
                 [, NUMBER=num] [, ACCESS=acc] [, FORM=frm]
                 [, RECL=rcl] [, NEXTREC=nxr] [, BLANK=blk])
```

All parameters except FILE=fn are optional.

FILE=fn
FILE=fn is required. fn is the reference to a file and must be preceded by FILE=. It is a character expression. Its value, when any trailing blanks are removed, must be 1 to 7 alphameric characters, the first one being alphabetic. It specifies the name of the file being inquired about and must be known to the program.

ERR=stn
stn is the number of a statement in the same program unit as the INQUIRE statement to which control is given when the value of fn (as described under FILE=fn) is not a valid file name.

IOSTAT=ios
ios is an integer variable or an integer array element. The value of ios is set positive if an error is detected; it is set to zero if no error is detected. VSAM return and reason codes are placed in ios.

EXIST=exs
exs is a logical variable or logical array element. It is assigned the value true if the file by the specified name exists; otherwise, it is assigned the value false.

THE FILE EXISTS: The following parameters have a value only if the file being inquired about exists; that is, exs has the value true. They are all optional.

OPENED=opn
opn is a logical variable or a logical array element. It is assigned the value true if the file specified is connected to a unit, otherwise, it is assigned the value false.

NAMED=nmd
nmd is a logical variable or a logical array element. If the file has a name (fn), nmd is assigned the value true; otherwise, it is assigned the value false.

NAME=nam
nam is a character variable or character array element. If the file has a name (fn), nam is assigned the value of name. name is not necessarily the same as the name in the FILE parameter (fn).

SEQUENTIAL=seq
seq is a character variable or a character array element. It is assigned the value YES if the file can be connected for sequential access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for sequential access.

DIRECT=dir
dir is a character variable or a character array element. It is assigned the value YES if the file can be connected for direct access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for direct access.

FORMATTED=fmt
fmt is a character variable or character array element. It is assigned the value YES if the file can be connected for formatted input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for formatted input/output.

UNFORMATTED=unf
unf is a character variable or character array element. It is assigned the value YES if the file can be connected for unformatted input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for unformatted input/output.

THE FILE IS CONNECTED TO AN EXISTING UNIT: The following parameters have a value only if the file exists (exs has the value true) and if the file is connected to a unit (opn has the value true). They are all optional.

NUMBER=num
num is an integer variable or integer array element. It is assigned the value of the external unit connected to the file.

ACCESS=acc
acc is a character variable or character array element. If there is a name fn, acc is assigned a value (SEQUENTIAL or DIRECT) associated with the connection of the external file.

FORM=frm
frm is a character variable or character array element. It is assigned the value FORMATTED if the file is connected for formatted input/output; UNFORMATTED if the file is connected for unformatted input/output.

THE FILE IS CONNECTED FOR DIRECT ACCESS I/O: The following parameters have a value only if the file exists (exs has the value true) and if the file is connected for direct access (acc=DIRECT). They are all optional. The file must have been explicitly opened.

RECL=rcl
rcl is an integer variable or integer array element. It is assigned the value of the record length of the file connected for direct access. The length is measured in characters for files consisting of formatted records and in bytes for files consisting of unformatted records.

NEXTREC=nxr
nxr is an integer variable or integer array element. It is assigned the value n+1, where n is the record number of the last record read or written on the direct access file. If the file is connected, but no records have been read or written since the connection, nxr is assigned the value 1.

THE FILE IS CONNECTED FOR FORMATTED I/O: The following parameter has a value only if the file exists (exs has the value true) and if the file is connected for formatted input/output (frm has the value FORMATTED). It is optional.

BLANK=blk
blk is a character variable or character array element. It is assigned the value NULL if blanks in arithmetic input fields are treated as blanks; ZERO if they are treated as zeros.

The parameters can be entered in any order. Each parameter cannot appear more than once in an INQUIRE statement. The same variable or array element may not be specified for more than one parameter in the same INQUIRE statement.

Valid INQUIRE Statement:

```
INQUIRE (FILE=DDNAME, IOSTAT=IOS, EXIST=LEX, OPENED=LOD,
         NAMED=LNMD, NAME=FN, SEQUENTIAL=SEQ, DIRECT=DIR,
         FORMATTED=FMT, UNFORMATTED=UNF, ACCESS=ACC, FORM=FRM,
         NUMBER=INUM, RECL=IRCL, NEXTREC=INR, BLANK=BLNK)
```

INQUIRE by Unit Number

This INQUIRE statement supplies information about an input/output unit.

A unit can be queried as to its existence and its connection to a file. If it is connected to a file, the inquiry is being made about the connection and the file connected. When this statement is executed, the unit specified by un may or may not be connected to a file. If the unit is connected to a file, the file may or may not exist. For example, an output unit may be connected to a file but no output has been written.

Syntax

```
INQUIRE ([UNIT=]un [, ERR=stn] [, IOSTAT=ios] [, EXIST=exs]
                   [, OPENED=opn] [, NAMED=nmd] [, NAME=nam]
                   [, SEQUENTIAL=seq] [, DIRECT=dir]
                   [, FORMATTED=fmt] [, UNFORMATTED=unf]
                   [, NUMBER=num] [, ACCESS=acc] [, FORM=frm]
                   [, RECL=rcl] [, NEXTREC=nxr] [, BLANK=blk]
```

All parameters except UNIT=un are optional.

UNIT=un
un is required. It is the reference to an I/O unit. un can be preceded optionally by UNIT=. It is an integer expression whose value represents the unit number that is being queried.

ERR=stn
stn is the number of a statement in the same program unit as the INQUIRE statement to which control is given when the value of un (as described under UNIT=un) is not a valid unit number.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is set positive if an error is detected; it is set to zero if no error is detected. VSAM return and reason codes are placed in ios.

EXIST=exs
exs is a logical variable or logical array element. It is assigned to value true if the specified unit exists and is known to the program unit. If neither of these conditions is met, exs is assigned the value false.

OPENED=opn
opn is a logical variable or logical array element. It is assigned the value true if the file specified is connected to a unit; otherwise, it is assigned the value false.

THE UNIT IS CONNECTED TO AN EXTERNAL FILE: The following parameters have a value only if the unit exists (exs has the value true) and the unit is connected to an external file (opn has the value true). They are all optional.

NAMED=nmd
nmd is a logical variable or a logical array element. It is assigned the value true if the file connected to the unit has a name; otherwise, it is assigned the value false.

NAME=nam
nam is a character variable or character array element. If the file connected to the unit has a name, it is assigned the value of the name of that file. If the file is unnamed, a default name is assigned.

SEQUENTIAL=seq
seq is a character variable or a character array element. It is assigned the value YES if the file can be connected for sequential access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for sequential access.

DIRECT=dir
dir is a character variable or a character array element. It is assigned the value YES if the file can be connected for direct access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for direct access.

FORMATTED=fmt
fmt is a character variable or character array element. It is assigned the value YES if the file can be connected for formatted input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for formatted input/output.

UNFORMATTED=unf
unf is a character variable or character array element. It is assigned the value YES if the file can be connected for formatted input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for formatted input/output.

NUMBER=num
num is an integer variable or integer array element. Its value is the value of un.

ACCESS=acc
acc is a character variable or character array element. It is assigned the value (SEQUENTIAL or DIRECT) associated with the connection of the external file.

FORM=frm
frm is a character variable or character array element. frm is assigned the value FORMATTED if the file is connected for formatted input/output; UNFORMATTED if the file is connected for unformatted output.

THE UNIT IS CONNECTED TO AN EXTERNAL FILE FOR DIRECT ACCESS I/O:
The following parameters have a value only if the unit exists (exs has the value true) and is connected to an external file for direct access input/output (acc has the value DIRECT). They are all optional.

RECL=rcl
rcl is an integer variable or integer array element. It is assigned the value of the record length of the direct access file. The length is measured in characters for files consisting of formatted records and in bytes for files consisting of unformatted records.

NEXTREC=nxr
nxr is an integer variable or integer array element. It is assigned the value n+1 where n is the record number of the last record read or written on the direct access file. If the file is connected, but no records have been read or written since the connection, nxr is assigned the value 1.

BLANK=blk
blk is a character variable or character array element. It is assigned the value NULL if blanks in arithmetic input fields are treated as blanks; ZERO if they are treated as zeros.

The parameters can be entered in any order unless UNIT=un is omitted. If omitted, un, as described under UNIT=un, must be first.

Valid INQUIRE Statement:

```
INQUIRE (0, IOSTAT=IACT(1), ERR=99999, EXIST=LACT(9),
         OPENED=LACT(8), NAMED=LACT(7), NAME=ACTUAL(1),
         SEQUENTIAL=ACTUAL(2), DIRECT=ACTUAL(3),
         FORMATTED=ACTUAL(4), UNFORMATTED=ACTUAL(5),
         ACCESS=ACTUAL(6), FORM=ACTUAL(7), NUMBER=IACT(2),
         RECL=IACT(3), NEXTREC=IACT(4), BLANK=ACTUAL(8))
```

INTEGER TYPE STATEMENT

See "Explicit Type Statement" on page 85.

INTRINSIC STATEMENT

The INTRINSIC statement identifies a name as representing a FORTRAN-supplied procedure (intrinsic function) (see "Appendix B. FORTRAN-Supplied Procedures" on page 204). This name can be a generic name or a specific name. See "Specific Names and Generic Names" on page 131. It also permits a specific intrinsic function name to be used as an actual argument.

Syntax

```
INTRINSIC name1 [, name2 ] ...
```

name
is the name of a VS FORTRAN intrinsic function.

The INTRINSIC statement is a specification statement and must precede statement function definitions and all executable statements.

Intrinsic functions are those functions known to the compiler. Intrinsic function names are either generic or specific. A generic name does not have a type unless it is also a specific name. When a generic name is used with any of the argument types available for that generic name, the specific named function corresponding to the argument type is chosen. This makes it unnecessary for the user to know which intrinsic function name goes with which argument type.

Appearance of a name in an INTRINSIC statement declares that name to be an intrinsic function name. If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit.

The following names of specific intrinsic functions must *not* be passed as actual arguments:

AMAX0	INT
AMAX1	LGE
AMIN0	LGT
AMIN1	LLE
CHAR	LLT
DMAX1	MAX0
DMIN1	MAX1
FLOAT	MIN0
ICHAR	MIN1
IDINT	REAL
IFIX	SNGL

IBM EXTENSION

CMPLX	QCMPLX
DBLE	QEXT
DBLEQ	QEXTD
DCMPLX	QFLOAT
DFLOAT	QMAX1
DREAL	QMIN1
HFIX	QREAL
IQINT	SNGLQ

END OF IBM EXTENSION

The appearance of a generic function name in an INTRINSIC statement does not cause the name to lose its generic property. Only one appearance of *name* in all of the INTRINSIC statements of a program unit is permitted. The same name must not appear in both an EXTERNAL and an INTRINSIC statement in a program unit.

If the name of a VS FORTRAN intrinsic function appears in an explicit specification statement, the type must confirm its associated type.

If the name of a FORTRAN intrinsic function appears in the dummy argument list of a subprogram, that name is not considered as the name of a FORTRAN intrinsic function in that program unit.

Specific Names and Generic Names

Generic names simplify referring to intrinsic functions because the same function name may be used with more than one type of argument (See "Appendix B. FORTRAN-Supplied Procedures" on page 204). Only a specific intrinsic function name may be used as an actual argument when the argument is an intrinsic function. For those intrinsic functions that require more than one argument, all arguments must be of the same type.

LOGICAL IF STATEMENT

See "IF Statements" on page 117.

LOGICAL TYPE STATEMENT

See "Explicit Type Statement" on page 85.

IBM EXTENSION

NAMELIST STATEMENT

The NAMELIST statement specifies one or more lists of names for use in READ and WRITE statements.

Syntax

```
NAMELIST /name1/ list1 /name2/ list2 ...
```

name
is a NAMELIST name. It is a name enclosed in slashes that must not be the same as a variable or array name.

list
is of the form a1, a2,..., an

where:

a
is a variable name or an array name.

The list of variables or array names belonging to a NAMELIST name ends with a new NAMELIST name enclosed in slashes or with the end of the NAMELIST statement. A variable name or an array name may belong to one or more NAMELIST lists.

Neither a dummy variable nor a dummy array name may appear in a NAMELIST list.

The NAMELIST statement must precede any statement function definitions and all executable statements. A NAMELIST name must be declared in a NAMELIST statement and may be declared only once. The name may appear only in input/output statements.

The NAMELIST statement declares a name name to refer to a particular list of variables or array names. Thereafter, the forms READ(un,name) and WRITE(un,name) are used to transmit data between the file associated with the unit un and the variables specified by the NAMELIST name name.

The rules for input/output conversion of NAMELIST data are the same as the rules for data conversion described in "General Rules for Data Conversion" on page 92 under "FORMAT Statement" on page 90. The NAMELIST data must be in a special form, described in "NAMELIST Input Data."

NAMELIST Input Data

Input data must be in a special form in order to be read using a NAMELIST list. The first character in each record to be read must be blank. The second character in the first record of a group of data records must be an ampersand (&) immediately followed by the NAMELIST name. The NAMELIST name must be followed by a blank and must not contain any embedded blanks. This name is followed by data items separated by commas. (A comma after the last item is optional.) The end of a data group is signaled by &END.

The form of the data items in an input record is:

- Name = Constant
  - The name may be an array element name or a variable name.
  - The constant may be integer, real, complex, logical, or character. (If the constants are logical, they may be

in the form T or .TRUE. and F or .FALSE., if the constants are characters, they must be included between apostrophes.)

- Subscripts must be integer constants.

- Array Name = Set of Constants (separated by commas)
  - The set of constants consists of constants of the type integer, real, complex, logical, or character.
  - The number of constants must be less than or equal to the number of elements in the array.
  - Successive occurrences of the same constant can be represented in the form c*constant, where c is a nonzero integer constant specifying the number of times the constant is to occur.

The variable names and array names specified in the input file must appear in the NAMELIST list, but the order is not significant. A name that has been made equivalent to a name in the input data cannot be substituted for that name in the NAMELIST list. The list can contain names of items in COMMON but must not contain dummy argument names.

Each data record must begin with a blank followed by a complete variable or array name or constant. Embedded blanks are not permitted in names or constants. Trailing blanks after integers and exponents are treated as zeros.

Examples:

All records have a blank in column 1.

```
          Column 2
          v
first card &NAM1 I(2,3)=5,J=4,B=3.2
                        .
                        .
                        .
last  card A(3)=4.0,L=2,3,7*4,&END
```

where NAM1 is defined in a NAMELIST statement as:

```
NAMELIST /NAM1/A,B,I,J,L
```

and assuming that A is a 3-element array and I and L are 3X3 element arrays.

NAMELIST Output Data

When output data is written using a NAMELIST list, it is written in a form that can be read using a NAMELIST list. All variable and array names specified in the NAMELIST list and their values are written out, each according to its type. Character data is included between apostrophes. The fields for the data are made large enough to contain all the significant digits. The values of a complete array are written out in columns.

Example:

```
NAMELIST /NAM1/A,B,I,J,L/NAM2/C,J,I,L
READ (CARD,NAM1)
WRITE (ITAPE,NAM1)
```

END OF IBM EXTENSION

OPEN STATEMENT

An OPEN statement may be used to:

- Connect an existing file to a unit.
- Create a file that is preconnected.
- Create a file and connect it to a unit.
- Change certain identifiers of a connection between a file and a unit.

Syntax

```
OPEN ( [UNIT=]un [, ERR=stn] [, STATUS=sta] [, FILE=fn]
       [, ACCESS=acc] [, BLANK=blk] [, FORM=frm]
       [, IOSTAT=ios] [, RECL=rcl] )
```

All parameters are optional except un.

UNIT=un
is required. It is the reference to an I/O unit. un can be preceded optionally by UNIT=. It is an integer expression whose value represents the unit number.

ERR=stn
stn is the number of a statement in the same program unit as the OPEN statement to which control is given when an error is detected during execution of the OPEN statement.

STATUS=sta
sta is a character expression. Its value when any trailing blanks are removed must be NEW, OLD, SCRATCH, or UNKNOWN. If STATUS is omitted, it is assumed to be UNKNOWN.

If the status of the external file is specified as:

- NEW, FILE=fn may be specified and the file fn must not exist.
- OLD, FILE=fn may be specified and the file fn must exist.
- SCRATCH, FILE=fn must not be specified and the file fn may or may not exist.
- UNKNOWN, FILE=fn is optional.

FILE=fn
fn is a character expression. Its value when any trailing blanks are removed is the name of the file to be connected to the unit specified by un. This file name must be a string of 1 to 7 alphameric characters, the first one being alphabetic.

ACCESS=acc
acc is a character expression whose value (when any trailing blanks are removed) must be SEQUENTIAL or DIRECT. It specifies the file as being accessed as a sequential or direct file. If ACCESS=acc is not specified, it is assumed to be SEQUENTIAL.

BLANK=blk
blk is a character expression whose value (when any trailing blanks are removed) must be NULL or ZERO. This specifier affects the processing of the arithmetic fields accessed by READ statements with format specification or with list-directed only. It is ignored for nonarithmetic fields,

for READ statements without format specification or with NAMELIST, and for all output statements. If NULL is specified, all blank characters in arithmetic formatted input fields on the specified unit are ignored, except that a field of all blanks has a value of zero. If ZERO is specified, all blanks, other than leading blanks, are treated as zeros. If this specifier is omitted and FORM=FORMATTED, a value of NULL is assumed.

FORM=frm
frm is a character expression whose value (when any trailing blanks are removed) must be FORMATTED or UNFORMATTED. This specifier indicates that the external file is connected for formatted or unformatted input/output. If this specifier is omitted for a file connected with direct access, a value of UNFORMATTED is assumed. If this specifier is omitted for a file connected with sequential access, a value of FORMATTED is assumed.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is set positive if an error is detected; it is set to zero if no error is detected. VSAM return and reason codes are placed in ios.

DIRECT ACCESS FILES: The following specifier is used with direct access files.

RECL=rcl
rcl is an integer expression. It is assigned the value of the record length of the file connected for direct access. The length is measured in characters for files consisting of formatted records and in bytes for files consisting of unformatted records.

Each of the parameters of the OPEN statement may appear only once. The unit specifier (un) must appear. All value assignments are made according to the rules for assignment statements.

If UNIT= is not specified, un must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, the parameters may appear in any order.

Before the OPEN statement is executed, the I/O unit specified by un may be either connected or not connected to an external file.

OPEN is required for direct-access and VSAM files. It is optional for sequential files.

It is invalid for internal files.

The I/O Unit is Not Connected to the External File

Successful execution of the OPEN statement connects the I/O unit specified by un to the external file specified by fn with the parameters specified (or assumed) in the OPEN statement. (See VS FORTRAN Application Programming: Guide for the parameters allowed with the different definitions of data sets.)

The I/O Unit is Connected to the External File

A unit connected in any program unit of an executable program is available in any other program unit of the executable program.

The unit reference and the file name are un and fn in the OPEN statement.

BEFORE EXECUTION OF OPEN

- If some parameters are specified on the OPEN statement, they must match the attributes of the connection of file fn (except that BLANK may be different).
- The external file fn must not be connected to an I/O unit.
- The OPEN is executed as a CLOSE (UNIT=un, STATUS=UNKNOWN) followed by an OPEN with unit un and external file fn1.
- If any error is detected, the unit un stays connected to file fn.

AFTER SUCCESSFUL OPEN

- Unit un stays connected to file fn.
- The new value of the BLANK specifier comes into effect.
- File fn exists (exs has the value true).
- If it had the NEW attribute, it is changed to OLD.
- The other attributes stay unchanged.
- The file is not repositioned at the beginning.
- The unit un is connected to the external file fn1. The attributes of the connection are described in VS FORTRAN Application Programming: Guide.
- The unit reference and the file name are un1 and fn in the OPEN statement (un1 different from un). An error is detected and the unit un stays connected to file fn.

CONDITIONS THAT PREVENT EXECUTION OF OPEN: Any of the following conditions prevent execution of the OPEN statement:

- Invalid unit number specified, that is, un.
- Invalid file name specified, that is, fn.
- Invalid values of the specifiers in the OPEN statement.
  - OLD specified for a file that does not exist.
  - ACCESS, FORM, REC do not match the actual attributes of an existing file.
  - RECL=rcl value is not positive integer.
  - OPEN statement specifies a different unit than the one the file is connected to.

Control transfers to the statement specified in ERR=stn or, if ERR=stn is not specified, execution of the program is terminated.

Examples:

Open a New External File: The following statement would open a new external file.

```
DDNAME = 'DDNAME'
OPEN (UNIT=2*IN-10, IOSTAT=IOS, ERR=99999, FILE=DDNAME,
      STATUS=NEW, ACCESS='SEQU'//'ENTIAL  ', FORM=FORMAT,
      BLANK=ZERO)
```

Open an Old External File: The following statement would open an old external file.

```
OPEN (0, IOSTAT=IACT(1), FILE='DDNAME',STATUS='OLD',
     ACCESS='SEQUENTIAL', FORM='FORMATTED',
     BLANK='NULL')
```

Open a Preconnected, Nonexistent File: The following statement would open a preconnected, nonexisting file unknown for direct.

```
OPEN (IOSTAT=IACT(1), ERR=99999, STATUS=UNKNOWN,
     ACCESS='DIRECT', RECL=32, UNIT=IN+6)
```

PARAMETER STATEMENT

The parameter statement assigns a name to a constant.

Syntax

```
PARAMETER ( name1 = c1 [, name2 = c2 ] ...)
```

name
is the name of a specific constant in this program unit (even if it looks like a hexadecimal constant, for example, Z0ABC). The name must be defined only once in a PARAMETER statement of a program unit.

c
is a constant or a constant expression of type integer, real, complex, logical, or character.

Before using the PARAMETER statement, *name* must have been specified by the IMPLICIT statement or an explicit type statement. (Otherwise the predefined conventions are used.)

The type and length of a name of a constant must not be changed by subsequent specification statements, including IMPLICIT statements. The following is *invalid*:

```
PARAMETER   (INT=10)

IMPLICIT    CHARACTER*5(I)
```

If the length of a character constant represented by a name has been explicitly specified previously or has been been specified as an asterisk, the length is considered to be the length of the value of the character expression (*c*).

If the name (*name*) is of type integer, real, or complex, the corresponding expression (*c*) must be a constant, the name of a constant, or another expression enclosed in parentheses. The exponentiation operator is not permitted unless the exponent is of type integer.

If the name (*name*) is of type character, the corresponding expression (*c*) must be a character expression containing only character constants or names of character constants. The expression result cannot exceed 255 characters in length.

If the name (*name*) is of type logical, the corresponding expression (*c*) must be a logical expression containing only logical constants or names of logical constants.

Each (*name*) is the name of a constant that becomes defined with the value of the expression (*c*) that appears to the right of the equal sign. The value assigned is determined by the rules used for assignment statements (see Figure 19 and Figure 20).

Any name of a constant that appears in an expression (*c*) must be defined by appearing previously on the left of an equal sign in the same or a preceding PARAMETER statement in the same program unit. If it is in the same PARAMETER statement, it must appear to the left of its usage.

Once defined, the name can be used in a subsequent expression or a DATA statement instead of the constant it represents. It must not be part of a FORMAT statement or a format specification.

The name of a constant must not be used to form part of another constant; for example, any part of a complex constant.

PAUSE STATEMENT

The PAUSE statement temporarily halts the execution of the object program and may display a message.

Syntax

```
PAUSE [n]
PAUSE ['message']
```

n
a string of 1 through 5 decimal digits.

'message'
a character constant enclosed in apostrophes and containing alphameric and/or special characters. Within the literal, an apostrophe is indicated by two successive apostrophes.

If either n or 'message' is specified, PAUSE displays the requested information. The program waits until operator intervention causes it to resume execution, starting with the next statement after the PAUSE statement or the next iteration of the DO loop, if it is the last statement of a DO range. For further information, see VS FORTRAN Application Programming: Guide.

PRINT STATEMENT

The PRINT statement transfers data from internal storage to an external device.

Syntax

```
PRINT fmt [,list]
```

fmt
can be one of the following:

- A statement number
- An integer variable
- A character constant
- A character array element
- A character array name
- A character expression

IBM EXTENSION

- An array name

END OF IBM EXTENSION

- An asterisk that indicates that printing is to be performed according to the data transmission rules of list-directed WRITE.

See "WRITE Statement—Formatted with Direct Access" on page 181 for explanations of these format identifiers.

list
is a list of output items and implied DO lists. An output list item can be:

- A variable name
- An array element
- A character substring
- An array name
- Any expression (except a character expression involving concatenation of operands whose length specification is an asterisk)

For a discussion of Implied DO lists, see "Implied DO in an Input/Output Statement" on page 74.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

If list is omitted, a blank record is transmitted to the output device unless the FORMAT statement referred to contains, as its first specification, a character constant or slashes. In this case, the record (or records) indicated by these specifications are transmitted to the output device.

PRINT fmt has the same effect as a WRITE (un,fmt) list where fmt and list are defined as above, and the value of un is installation dependent. See "WRITE Statement—Formatted with Sequential Access" on page 185.

Valid PRINT Statement:

```
PRINT*,EIGHT8
```

PROGRAM STATEMENT

The PROGRAM statement assigns a name to a main program. It must be the first statement in the main program.

Syntax

```
PROGRAM name
```

name
is the name of the main program in which this statement appears.

A main program cannot contain any BLOCK DATA, SUBROUTINE, FUNCTION, or ENTRY statements.

IBM EXTENSION

A RETURN statement may appear; it has the same effect as a STOP statement.

END OF IBM EXTENSION

The PROGRAM statement can only be used in a main program but is not required. If it is used, it must be the first statement of the main program. If it is not used, the name of the main program is assumed by this compiler to be MAIN.

The name must not be the same as any other name in the main program or as the name of a subprogram or common block in the same executable program. The name of a program does not have any type and the other specification statements have no effect on this name.

Execution of a program begins with the execution of the first executable statement of the main program. A main program may not be referred to from a subprogram or from itself.

The READ statements transfer data from an external device to storage or from one internal file to another.

Forms of the READ Statement:

IBM EXTENSION

1. READ Statement—Asynchronous

END OF IBM EXTENSION

2. READ Statement—Formatted with Direct Access
3. READ Statement—Formatted with Sequential Access
4. READ Statement—Unformatted with Direct Access
5. READ Statement—Unformatted with Sequential Access
6. READ Statement with Internal Files
7. READ Statement with List-Directed I/O

IBM EXTENSION

8. READ Statement with NAMELIST

END OF IBM EXTENSION

IBM EXTENSION

READ Statement—Asynchronous

The asynchronous READ statement transmits unformatted sequential data between direct access or sequential storage devices. The asynchronous READ statement provides high-speed input. The statements are asynchronous in that while data transfer is taking place, other program statements may be executed. An OPEN statement is not permitted for asynchronous I/O. The unit and statement identifier are the only items allowed within the parentheses.

Syntax

```
READ ( [UNIT=]un, ID=id ) [list]
```

UNIT=un
un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression of length 4. It is the reference to an I/O unit.

ID=id
id is an integer constant or integer expression of length 4. It is the identifier for the READ statement.

list
is an asynchronous I/O list and may have any of four forms:

```
e
e1...e2
e1...
...e2
```

where:

e
is the name of an array.

e1 and e2
are the names of elements in the same array. The ellipsis (...) is an integral part of the syntax of the list and must appear in the positions indicated.

The unit specified by un must be connected to a file that resides on a sequential or direct-access device. The array (e) or array elements (e1 through e2) constitute the receiving area for the data to be read.

The asynchronous READ statement initiates a transmission. The WAIT statement, that must be executed for each asynchronous READ, terminates the transmission cycle. When executed after an asynchronous READ, the WAIT statement enables the program to refer to the transmitted data. This process ensures that a program will not refer to a data field while transmission to it is still in progress.

The asynchronous READ statement differs from other READ statements in that a special parameter, ID=id, is specified within the parentheses of the statement. ID=id establishes a unique identification for the READ statement.

Synchronous READ statements may be executed for the file only after all asynchronous READ and WRITE operations have been completed and a REWIND has been executed for the file. Conversely, asynchronous READ statements may be executed for a file previously read synchronously after a REWIND or CLOSE has been executed.

Execution of an asynchronous READ statement initiates reading of the next record on the specified file. The record may contain more or less data than there are bytes in the receiving area. If there is more data, the excess is not transmitted to the receiving area; if there is less, the values of the excess array elements remain unaltered. The extent of the receiving area is determined as follows:

- If e is specified, the entire array is the receiving area.
- If e1...e2 is specified, the receiving area begins at array element e1 and includes every element up to and including e2. The subscript value of e1 must not exceed that of e2.
- If e1... is specified, the receiving area begins at element e1 and includes every element up to and including the last element of the array.
- If ...e2 is specified, the receiving area begins at the first element of the array and includes every element up to and including e2.

If list is not specified, there is no receiving area, no data is transmitted, and a record is skipped.

Subscripts in the list of the asynchronous READ must not be defined as array elements in the receiving area. If a function reference is used in a subscript, the function reference may not perform I/O on any file.

Given an array with elements of length len, transmission begins with the first len bytes of the record being placed in the first specified (or implied) array element. Each successive len bytes of the record are placed in the array element with the next highest subscript value. Transmission ceases after all elements of the receiving area have been filled, or the entire record has been transmitted—whichever occurs first. If the record length is less than the receiving area size, the last array element to receive data may receive fewer than len bytes.

The specified array may be multidimensional. Array elements are filled sequentially. Thus, during transmission, the leftmost subscript quantity increases most rapidly, and the rightmost least rapidly.

Any number of program statements may be executed between an asynchronous READ and its corresponding WAIT, subject to the following rules:

- No array element in the receiving area may appear in any such statement. This and the following rules apply also to indirect references to such array elements; that is, reference to or redefinition of any variable or array element associated by COMMON or EQUIVALENCE statements, or argument association with an array element in the receiving area.
- No executable statement may appear that redefines or undefines a variable or array element appearing in the subscript of e1 or e2. See "Valid and Invalid VS FORTRAN Programs" on page 3.
- If a function reference appears in the subscript expression of e1 or e2, the function may not be referred to by any statements executed between the asynchronous READ and the corresponding WAIT. Also, no subroutines or functions may be referred to that directly or indirectly refer to the function in the subscript reference, or to which the subscript function directly or indirectly refers.
- No function or subroutine may be executed that performs input or output on the file being manipulated, or that

contains object-time dimensions that are in the receiving area (whether they be dummy arguments or in a common block).

Valid READ Statement:

```
READ (ID=10, UNIT=3*IN-3) ACTUAL(3)...ACTUAL(7)
```

END OF IBM EXTENSION

READ Statement—Formatted with Direct Access

This READ statement transfers data from an external direct-access device into internal storage. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The data must reside on an external file that is connected for direct access to a unit (see "OPEN Statement" on page 134).

Syntax

```
READ ( [UNIT=]un, [FMT=]fmt, REC=rec [, ERR=stn]
       [, IOSTAT=ios] ) [list]
```

UNIT=un
un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression of length 4. It is the reference to an I/O unit.

If UNIT= is included, FMT= must be used. If UNIT= is not included, the unit reference number must appear first.

FMT=fmt
fmt is a required format identifier. It can optionally be preceded by FMT=.

If FMT= is not included, the format identifier must appear second.

If both UNIT= and FMT= are included, all the parameters, except list, can appear in any order.

The format identifier (fmt) can be:

- A statement number
- An integer variable
- A character constant
- A character array element
- A character array name
- A character expression

IBM EXTENSION

- An array name

END OF IBM EXTENSION

The statement number must be the statement number of a FORMAT statement in the same program unit as the READ statement.

The integer variable must have been initialized by an ASSIGN statement with the number of a FORMAT statement. The FORMAT statement must be in the same program unit as the READ statement.

The character constant must constitute a valid format. The constant must be delimited by apostrophes, must begin with a left parenthesis, and end with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. An apostrophe in a constant enclosed in apostrophes is represented by two consecutive apostrophes.

The character array element must contain character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. Blank characters may precede the left parenthesis and character data may follow the right

parenthesis. The length of the format identifier must not exceed the length of the array element.

The character array name must contain character data whose leftmost characters constitute a valid format identifier. The length of the format identifier may exceed the length of the first element of the array; it is considered the concatenation of all the array elements of the array in the order given by array element ordering.

IBM EXTENSION

The array name may be of type integer, real, double precision, logical, or complex.

The data must be a valid format identifier as described under character array name above.

END OF IBM EXTENSION

The character expression may contain concatenations of character constants, character array elements and character array names. Its value must be a valid format identifier. The operands of the expression must have length specifications that contain only integer constants or names of integer constants. (See "VS FORTRAN Expressions" on page 25.)

REC=rec
rec is a relative record number. It is an integer expression whose value must be greater than zero. It represents the relative position of a record within the external file associated with un. The relative record number of the first record is 1.

ERR=stn
stn is the number of a statement in the same program unit as the READ statement. Transfer is made to stn if an error is detected.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is detected; and zero if no error is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list. It can contain variable names, array element names, character substring names, array names, and implied DO lists. See "Implied DO in an Input/Output Statement" on page 74.

An item in the list, or an item associated with it through EQUIVALENCE, COMMON, or argument passing, must not contain any portion of the format identifier fmt.

Valid READ statements:

```
READ (un,fmt,REC=rec) list

READ (un,FMT=fmt,REC=rec) list

READ (UNIT=un,FMT=fmt,REC=rec) list

READ (REC=rec,FMT=fmt,UNIT=un)
```

Invalid READ statements:

READ (*fmt*,*un*,REC=*rec*)	*un* must appear before *fmt*.
READ (FMT=*fmt*,*un*,REC=*rec*) *list*	*un* must appear first because UNIT= is not included.
READ (b,UNIT=*un*,REC=*rec*) *list*	FMT must be used because UNIT= is included.
READ (*un*,*fmt*) *list*	REC=*rec* must be specified for direct-access.

If this READ statement is encountered, the unit specified must exist and the file must be connected for direct access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

This statement permits a programmer to read records randomly from any location within an external file. It contrasts with the sequential input statements that process records, one after the other, from the beginning of an external file to its end. With the direct-access statements, a programmer can go directly to any record in the external file, process a record and go directly to any other record without having to process all the records in between.

Each record in a direct-access file has a unique number associated with it. This number is the same as specified when the record is written. The programmer must specify in the READ statement not only the unit reference number, but also the number of the record to be read. Specifying the record number permits operations to be performed on selected records of the file instead of on records in their sequential order.

The OPEN statement specifies the size and the type of the records in the direct-access file. All the records of a file connected for direct access have the same length.

DATA TRANSMISSION: A READ statement with FORMAT starts data transmission at the beginning of the record specified by REC=*rec*. The format codes in the format identifier *fmt* are taken one by one and associated with every item of the list in the order they are specified. The number of character data specified by the format code is taken from the record, converted according to the format code and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item of the list or when the end of the record specified by *rec* is reached.

If the list is not specified and the format identifier starts with an I, E, F, D, G, or L format code, or is empty (that is, FORMAT()), the internal record number is increased by one but no data is transferred.

IBM EXTENSION

VS FORTRAN adds that, if the format identifier starts with a Q or Z format code, the internal record number is increased by one but no data is transferred.

END OF IBM EXTENSION

DATA AND I/O LIST: The length of every FORTRAN record is specified in RECL of the OPEN statement. If the record *rec* contains *more* data than is necessary to satisfy all the items of the list and the associated format identifier, the remaining data is ignored. If the record *rec* contains *less* data than is necessary to satisfy all the items of the list and the associated format identifier, an error is detected. If the format identifier indicates (for

example, slash format code) that data be taken from the next record, then the internal record number rec is increased by one and data transmission continues with the next record. The INQUIRE statement can be used to determine the record number.

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be read, only that an error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to ios when an error is detected. If ERR is specified, then execution continues with the statement specified with the ERR, if present, or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, program execution is terminated when an error is detected.

Valid READ Statement:

```
READ (UNIT=2*IN-10, FMT='(I9)', REC=3)
```

READ Statement—Formatted with Sequential Access

This READ statement transfers data from an external I/O device to storage. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The data must reside in an external file that is connected for sequential access to a unit. (See "OPEN Statement" on page 134.)

The sequential I/O statements with format identifiers process records one after the other from the beginning of an external file to its end.

Syntax

```
READ ( [UNIT=]un, [FMT=]fmt [, ERR=stn] [, END=stn]
       [, IOSTAT=ios] ) [list]
READ fmt [, list]
```

UNIT=un

un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression or an asterisk (*). It is the reference to an I/O unit.

If UNIT= is included, FMT= must be used and all the parameters can appear in any order.

If UNIT= is not included, un must appear first in the statement. The other parameters may appear in any order.

In the form of the READ where un is not specified, un is installation dependent.

FMT=fmt

fmt is a required format identifier. It can optionally be preceded by FMT=.

If FMT= is not included, the format identifier must appear second.

If both UNIT= and FMT= are included, all the parameters, except list, can appear in any order.

The format identifier (fmt) can be:

- A statement number
- An integer variable
- A character constant
- A character array element
- A character array name
- A character expression

IBM EXTENSION

- An array name

END OF IBM EXTENSION

See "READ Statement—Formatted with Direct Access" on page 146 for explanations of these format identifiers.

ERR=stn

stn is the number of an executable statement in the program unit containing the READ statement. Transfer is made to stn if an error is detected.

END=stn

is the number of an executable statement in the program unit containing the READ statement. Transfer is made to stn when

the end of the external file is encountered.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is detected; and zero if no error is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list. It can contain variable names, array element names, character substring names, array names, and implied DO lists. See "Implied DO in an Input/Output Statement" on page 74. In the form of the READ where un is not specified, if the list is not present, the comma must be omitted. An item in the list, or an item associated with it through EQUIVALENCE, COMMON or argument passing, must not contain any portion of the format identifier fmt.

Valid READ statements:

```
READ (un,fmt) list

READ (un, FMT=fmt) list

READ (UNIT=un, FMT=fmt) list    FMT=fmt can appear first.

READ fmt, list

READ (5,98) A,B,(C(I,K),I=1,10)

READ (IOSTAT=IOS, UNIT=2*IN-10, FMT='(I9)', END=3600)
```

Invalid READ Statements:

READ (fmt,un)	un must appear before fmt.
READ (FMT=fmt, un) list	un must appear first because UNIT= is not included.
READ (fmt, UNIT=un) list	FMT must be used because UNIT= is included.
READ FMT=fmt, list	FMT must not be used in this form of READ.

If this READ statement is encountered, the unit specified must exist and the file must be connected for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

DATA TRANSMISSION: A READ statement with FORMAT starts data transmission at the beginning of a record. The format codes in the format identifier fmt are taken one by one and associated with every item of the list in the order they are specified. The number of character data specified by the format code is taken from the record, converted according to the format code, and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item of the list or when the end of file is reached.

DATA AND I/O LIST: If the record contains more data than is necessary to satisfy all the items of the list and the associated format specification, the extra data is skipped over. The next READ statement with FORMAT will start with the next record if no other I/O statement is executed on that file. If the record contains less data than is necessary to satisfy all the items of the list and the associated format identifier, an error is detected.

If the list is not specified and the format identifier starts with an I, E, F, D, G, or L format code or is empty (that is, FORMAT()), a record is skipped over.

IBM EXTENSION

VS FORTRAN adds the Q and Z format codes to the list.

END OF IBM EXTENSION

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be read, only that an error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to ios when an error is detected. Then execution continues with the statement specified with the ERR, if present, or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, object program execution is terminated when an error is detected.

END OF FILE: Transfer is made to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read into before the end of the file was encountered. If IOSTAT=ios is specified, a negative integer value is assigned to ios. Then execution continues with the statement specified with END, if present, or with the next statement if END is not specified. If END and IOSTAT are both omitted, object program execution is terminated when the end of the file is encountered.

READ Statement—Unformatted with Direct Access

This statement transfers data without conversion from an external direct-access I/O device into internal storage. The data must reside on an external file that is connected with direct access to a unit (see "OPEN Statement" on page 134).

Syntax

```
READ ( [UNIT=]un, REC=rec [, ERR=stn] [, IOSTAT=ios] )
       [list]
```

UNIT=un
un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression of length 4. It is the reference to an I/O unit.

If UNIT= is not included, un must appear first in the statement. The other parameters may appear in any order.

If UNIT= is included, all the parameters can appear in any order.

REC=rec
rec is a relative record number. It is an integer expression whose value must be greater than zero. It represents the relative position of a record within the external file associated with un. The relative record number of the first record is 1.

ERR=stn
stn is the number of a statement in the same program unit as the READ statement. Transfer is made to stn if an error is detected.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list and can contain variable names, array elements, character substring names, array names, and implied DO lists. See "Implied DO in an Input/Output Statement" on page 74.

Valid READ statements:

```
READ (un,REC=rec) list

READ (REC=rec, UNIT=un)

READ (IOSTAT=IOS, UNIT=11, REC=3) ACTUAL(3)(1:)

READ (IOSTAT=IACT(1),UNIT=3*IN-2,FMT=*) ACTUAL(1)
```

Invalid READ statements:

`READ (REC=rec,un) list`	UNIT must be used because un is after REC=rec.
`READ (UNIT=un) list`	REC=rec must be specified for direct files.

If this READ statement is encountered, the unit must exist and the file must be connected for direct access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to

a default file name. If the file is not preconnected, an error is detected.

DATA TRANSMISSION: A READ statement without format starts data transmission at the beginning of the record specified by REC=rec. The number of character data specified by the type of each item in the list is taken from the record and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item of the list.

If the list is not specified, the internal record number is increased by one but no data is transferred. The INQUIRE statement can be used to determine the record number.

DATA AND I/O LIST: The length of the FORTRAN records in the file are specified by RECL in the OPEN statement. If the record rec contains more data than is necessary to satisfy all the items of the list, the extra data is ignored. If the record rec contains less data than is necessary to satisfy all the items of the list, an error is detected.

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be read, only that an error occurred during transmission of data. If IOSTAT=ios is specified, a positive integer value is assigned to ios when an error is detected. Then execution continues with the statement specified with ERR, if present, or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, program execution is terminated when an error is detected.

READ Statement—Unformatted with Sequential Access

This READ statement transfers data without conversion from an external I/O device into internal storage. The data resides on an external file that is connected for sequential access to a unit (see "OPEN Statement" on page 134).

The sequential I/O statements without format control process records one after the other from the beginning of an external file to its end.

The ENDFILE, REWIND, and BACKSPACE statements may be used to manipulate the file.

Syntax

```
READ ( [UNIT=]un [, ERR=stn] [, END=stn] [, IOSTAT=ios] )
       [list]
```

UNIT=un
un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression of length 4 (or an asterisk (*)). It is the reference to an I/O unit. An asterisk (*) represents an installation-dependent unit.

If UNIT= is not included, un must appear first in the statement. The other parameters may appear in any order.

If UNIT= is included, all the parameters can appear in any order.

ERR=stn
stn is the number of a statement in the same program unit as the READ statement. Transfer is made to stn if an error is detected.

END=stn
is the number of an executable statement in the program unit containing the READ statement. Transfer is made to stn when the end of the external file is encountered.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list. It can contain variable names, array element names, character substring names, array names, and implied DO lists. See "Implied DO in an Input/Output Statement" on page 74.

Valid READ statements:

```
READ (un) list

READ (UNIT=un) list

READ (un)

READ (IOSTAT=IOS, UNIT=11)
```

Invalid READ statements:

```
READ un, list          un must be in parentheses.

READ, list             (un) must be included.
```

If this READ statement is encountered, the unit specified by un must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

DATA TRANSMISSION: A READ statement without conversion starts data transmission at the beginning of a record. The data specified by the item in the list is taken from the record and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item of the list or when the end of file is reached.

If the list is not specified, a record is passed over without transmitting any data.

DATA AND I/O LIST: If the record contains more data than is necessary to satisfy all the items of the list, the extra data is skipped over. The next READ statement without format will start with the next record if no other I/O statement is executed on that file. If the record contains less data than is necessary to satisfy the list, an error is detected.

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be read, only that an error occurred during transmission of data. If IOSTAT=ios is specified, a positive integer value is assigned to ios when an error is detected. Then execution continues with the statement specified with ERR, if present or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, program execution is terminated when an error is detected.

END OF FILE: Transfer is made to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read into before the end of the file was encountered. If IOSTAT=ios is specified, a negative integer value is assigned to ios when an end of file is detected. Then execution continues with the statement specified with END if present, or with the next statement if END is not specified. If END and IOSTAT are both omitted, program execution is terminated when the end of the file is encountered.

READ Statement with Internal Files

This READ statement transfers data from one area of internal storage into another area of internal storage. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The area in internal storage that is read from is called an internal file.

An internal file is always

- Connected to a unit
- Positioned before data transmission at the beginning of the storage area represented by the unit identifier
- Accessed sequentially with a FORMAT statement (see "FORMAT Statement" on page 90)

Syntax

```
READ ( [UNIT=]un, [FMT=]fmt [, ERR=stn] [, END=stn]
       [, IOSTAT=ios] ) [list]
```

UNIT=un
un is the reference to an area of internal storage called an internal file. It can optionally be preceded by UNIT=. It can be the name of:

A character variable
A character array
A character array element
A character substring

If UNIT= is included, FMT= must be used. If UNIT= is not included, the unit reference must appear first.

FMT=fmt
fmt is a required format identifier. It can optionally be preceded by FMT=.

The format identifier can be:

A statement number
An integer variable
A character constant
A character array element
A character expression

IBM EXTENSION

An array name

END OF IBM EXTENSION

See "READ Statement—Formatted with Direct Access" on page 146 for explanations of these format identifiers.

The format specification must not be:

- In the area un.
- Associated with un through EQUIVALENCE, COMMON or argument passing.

If FMT= is not included, the format specification must appear second.

If both UNIT= and FMT= are included, all the parameters, except list, can appear in any order.

ERR=stn
stn is the number of a statement in the same program unit as the READ statement. Transfer is made to stn if an error is detected.

END=stn
is the number of an executable statement in the program unit containing the READ statement. Transfer is made to stn when the end of the storage area (un) is encountered.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list and can contain variable names, array elements, character substring names, array names, and implied DO lists. See "Implied DO in an Input/Output Statement" on page 74.

An item in the list must not be:

- Contained in the area represented by un
- Associated with any part of un through EQUIVALENCE, COMMON, or argument passing

Valid READ statements:

READ (un,fmt) list

READ (un,FMT=fmt) list

READ (UNIT=un,FMT=fmt) list

Invalid READ statements:

READ (fmt,un) list	un must appear before fmt.
READ (FMT=fmt,un) list	un must appear first because UNIT= is not included.
READ (fmt,UNIT=un) list	FMT must be used because UNIT= is included.

DATA TRANSMISSION: An internal READ statement starts data transmission at the beginning of the storage area specified by un. The format codes in the format specification fmt are taken one by one and associated with every item of the list in the order they are specified. The number of character data specified by a format code is taken from the storage area un, converted according to the format code, and moved into the storage associated with the corresponding item in the list. Data transmission stops when data has been moved to every item of the list or when the end of the storage area a is reached.

If un is a character variable, a character array element name, or a character substring name, it is treated as one record only in relation to the format identifier.

If un is a character array name, each array element is treated as one record in relation to the format identifier.

DATA AND I/O LIST: The length of a record is the length of the character variable, character substring name, character array element specified by un when the READ statement is executed.

If a record contains more data than is necessary to satisfy all the items in the list and the associated format identifier, the remaining data is ignored.

If a record contains less data than is necessary to satisfy all the items in the list and the associated format identifier, an error is detected.

If the format identifier indicates (for example, slash format code) that data be moved from after the character variable, character substring, or the last array element of a character array, an end of file is detected. If it is not the last array element in the character array, data is taken from the next array element.

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be read, only that an error occurred during transmission of datun. If IOSTAT=ios is specified, a positive integer value is assigned to ios when an error is detected. Then execution continues with the statement specified with ERR if present or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, program execution is terminated when an error is detected.

END OF FILE: Transfer is made to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read into before the end of the file was encountered. If IOSTAT=ios is specified, a negative integer value is assigned to ios when an end of file is detected. Then execution continues with the statement specified with END if present or with the next statement if END is not specified. If END and IOSTAT are both omitted, program execution is terminated when the end of the file is encountered.

Example:

```
  1   CHARACTER* 120 CHARVR

  2   READ (UNIT=5, FMT=100) CHARVR

100   FORMAT (A120)

  3   ASSIGN 200 TO J

  4   IF (CHARVR (3:4).EQ. 'AB') ASSIGN 300 TO J

  5   READ(UNIT = CHARVR, FMT=J) A1, A2, A3

200   FORMAT(4X,F5.1, F10.3, 3X, F12.8)

300   FORMAT (4X, F3.1, F6.3, 20X, F8.4)
```

Statement 1 defines a character variable, CHARVR, of fixed length 120. Statement 2 reads into CHARVR 120 characters of input. Statement 3 assigns the format number 200 to the integer variable J. Statement 4 tests the third and fourth characters of CHARVR to determine which type of input is to be processed. If these two characters are AB, then the format numbered 300 replaces the format numbered 200 and is used for processing the data. This is done by assigning 300 to the integer variable J. Statement 5 reads the file and performs the conversion using the appropriate FORMAT statement and assigning values to A1, A2, and A3.

READ Statement with List-Directed I/O

This statement transfers data from an external device into internal storage. The type of the items specified in this statement determines the conversion to be performed. The data resides on an external file that is connected for sequential access to a unit (see "OPEN Statement" on page 134).

Syntax

```
READ ( [UNIT=]un, [FMT=]* [, ERR=stn] [, END=stn]
       [, IOSTAT=ios] ) [list]
READ * [, list]
```

UNIT=un
un is required in the first form of the READ statement. It can optionally be preceded by UNIT=. un is an unsigned integer expression (or an asterisk (*)). It is the reference to an I/O unit. An asterisk (*) represents an installation-dependent unit.

If UNIT= is not included, un must appear first in the statement. The other parameters may appear in any order.

If UNIT= is included, all the parameters can appear in any order.

In the form of the READ where un is not specified, un is installation dependent.

FMT=*
specifies that a list-directed READ is to be executed. It can optionally be preceded by FMT=.

If FMT= is not included, the format identifier must appear second.

If both UNIT= and FMT= are included, all the parameters, except list, can appear in any order.

ERR=stn
stn is the number of a statement in the same program unit as the READ statement. Transfer is made to stn if an error is detected.

END=stn
is the number of an executable statement in the program unit containing the READ statement. Transfer is made to stn when the end of the external file is encountered.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list and can contain variable names, array element names, character substring names, array names, and implied DO lists. See "Implied DO in an Input/Output Statement" on page 74.

Valid READ statements:

```
READ (un,*) list
READ (un,FMT=*) list
READ (FMT=*,UNIT=un) list
READ (*,*) list
READ *, list
READ (IOSTAT=IACT(1), UNIT=3*IN-2, FMT=*) ACTUAL(1)
```

Invalid READ statements:

`READ (*,un) list`	un must appear before *.
`READ (FMT=*,un) list`	un must appear first because UNIT= is not included.
`READ (*,UNIT=un) list`	FMT must be used because UNIT= is included.
`READ FMT=*, list`	FMT must not be specified.

If this READ statement is encountered, the unit specified by un must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

DATA TRANSMISSION: A READ statement with list-directed I/O accessing an external file starts data transmission at the beginning of a record. One value on the external file is transferred to each item of the list in the order they are specified. The conversion to be performed depends on the type and length of the name of the item in the list. Data transmission stops when data has been transmitted to every item in the list, when a slash separator is encountered in the file or when the end of file is reached.

DATA AND I/O LIST: If the record contains more data than is necessary to satisfy all the items of the list, the extra data is skipped over. The next READ statement with list-directed I/O will start with the next record if no other I/O statement is executed on that file. If the record contains less data than is necessary to satisfy the list and the record does not have a slash after the last element, an error is detected. If the list has not been satisfied when a slash separator is found, the remaining items in the list remain unaltered and execution of the READ is terminated.

Transfer is made to the statement specified by ERR if an input error occurs. No indication is given of which record or records could not be read, only that an error occurred during transmission of data. If IOSTAT=ios is specified, a positive integer value is assigned to ios when an error is detected. Then execution continues with the statement specified with ERR, if present, or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, object program execution is terminated when an input error occurs.

END OF FILE: Transfer is made to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read before the end of the file was encountered. If IOSTAT=ios is specified, a negative integer value is assigned to ios when an end of file is detected. Then execution continues with the statement specified with END, if present, or with the next statement if END is not specified. If END and IOSTAT are both omitted, object program execution is terminated when the end of the file is encountered.

IBM EXTENSION

READ Statement with NAMELIST

This statement transfers data from an external I/O device into storage. The type of the items specified in the NAMELIST determines the conversions to be performed. The data resides on an external file that is connected for sequential access to a unit (see "OPEN Statement" on page 134).

Syntax

```
READ ( un, name [, ERR=stn] [, END=stn] [, IOSTAT=ios] )
```

un
is required. un is an unsigned integer expression of length 4. It is the reference to an I/O unit.

name
is a NAMELIST name. This name must appear as the second parameter in the list and must be the same as the name in a NAMELIST statement that precedes the READ statement (see "NAMELIST Statement" on page 132).

ERR=stn
stn is the number of a statement in the same program unit as the READ statement. Transfer is made to stn if an error is detected.

END=stn
is the number of an executable statement in the program unit containing the READ statement. Transfer is made to stn when the end of the external file is encountered.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

Valid READ statements:

```
READ (un,name)

READ (IN+IN+3, NAMEIN, IOSTAT=IOS)
```

Invalid READ statements:

```
READ (name,un)          un must appear before name.

READ (un,name) list     list must not be specified.
```

If this READ statement is encountered, the unit specified by un must exist and it must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

The NAMELIST I/O statements associate the name given to the data in the FORTRAN program with the data itself. There is no format identifier but the data is converted according to the type of data in the FORTRAN program. The data on the external file must be in a specific format. See "NAMELIST Input Data" on page 132.

The READ statement specifies the list of data to be transferred by referring to a NAMELIST statement. This form of data transmission is useful for debugging purposes.

BACKSPACE and REWIND should not be used with NAMELIST I/O. If they are, the results are unpredictable (see "BACKSPACE Statement" on page 54 and "REWIND Statement" on page 166).

DATA TRANSMISSION: A READ statement with NAMELIST starts data transmission from the beginning of the NAMELIST with name name on the external file. The names associated with the NAMELIST name name in the NAMELIST statement are matched with the names of the NAMELIST name on the external file. When a match is found, the value associated with the name on the external file is converted to the type of the name and transferred into storage. If a match is not found, an error is detected.

DATA AND NAMELIST: The NAMELIST name name must appear on the external file. The variable names or array names associated with the NAMELIST name name in the NAMELIST statement must appear on the external file. They are read in the order they are specified in the NAMELIST statement, but they can appear in any order on the external file (see "NAMELIST Input Data" on page 132 for the format of the input data).

Transfer is made to the statement specified by ERR if an input error occurs. No indication is given of which record or records could not be read, only that an error occurred during transmission of data. If ERR is omitted, program execution is terminated when an error occurs.

END OF FILE: Transfer is made to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read before the end of the file was encountered. If END is omitted, object program execution is terminated when the end of the file is encountered.

END OF IBM EXTENSION

REAL TYPE STATEMENT

See "Explicit Type Statement" on page 85.

RETURN STATEMENT

The RETURN statement returns control to a calling program.

IBM EXTENSION

In a main program, a RETURN statement performs the same function as a STOP statement.

END OF IBM EXTENSION

The RETURN statement can be used in either a **function** or a **subroutine** subprogram.

RETURN Statement in a Function Subprogram

Function subprograms may contain RETURN statements. The RETURN statement signifies a logical conclusion of the computation and returns the computed function value and control to the calling program. See also "FUNCTION Statement" on page 111.

Syntax

```
RETURN
```

Execution of a RETURN statement terminates the association between the dummy arguments of the subprogram and the current actual arguments. All entities (that is, common blocks, variables, or arrays) within the subprogram become undefined except:

- Entities specified in SAVE statements (see "SAVE Statement" on page 168)
- Entities given an initial value in a DATA or explicit specification statement and whose initial values were not changed
- Entities in blank common
- Entities in named common that appear in the subprogram and appear in at least one other program unit that is referring either directly or indirectly to the subprogram

All variables that are defined with a statement number become undefined regardless of whether the variable is in common or specified in a SAVE statement.

A function subprogram must not be referred to twice during the execution of an executable program without the execution of a RETURN statement in that subprogram. (See also "END Statement" on page 77.)

RETURN Statement in a Subroutine Subprogram

Subroutine subprograms may contain RETURN statements. The RETURN statement signifies a logical conclusion of the computation and returns control to the calling program. See also "SUBROUTINE Statement" on page 173.

Syntax

```
RETURN  [m]
```

m
: is an integer expression. m must be within the range of the argument list. If it is not or if it is less than or equal to zero, the RETURN is executed, as if there were no m specified.

The normal sequence of execution following the RETURN statement of a subroutine subprogram is to the next statement following the CALL statement in the calling program. It is also possible to return to any numbered statement in the calling program by using a return of the type RETURN m.

Execution of a RETURN statement terminates the association between the dummy arguments of the subprogram and the current actual arguments. All entities within the subprogram become undefined except:

- Entities specified in SAVE statements. (See "SAVE Statement" on page 168.)
- Entities given an initial value in a DATA or explicit specification statement and where initial values were not changed.
- Entities in blank common.
- Entities in named common that appear in the subprogram and appear in at least one other program unit that is referring either directly or indirectly to the subprogram.

All variables that are defined with a statement number become undefined regardless of whether the variable is in common or specified in a SAVE statement.

A subprogram must not be referred to twice during the execution of an executable program without the execution of a RETURN statement in that subprogram.

A CALL statement that is used with a RETURN m form may be best understood by comparing it to a CALL and computed GO TO statement in sequence. For example, the following CALL statement:

```
CALL SUB (P,*20,Q,*35,R,*22)
```

is equivalent to:

```
CALL SUB (P,Q,R,I)
GO TO (20,35,22),I
```

where the index I is assigned a value of 1, 2, or 3 in the called subprogram.

REWIND STATEMENT

The REWIND statement positions an external file at the beginning of the first record of the file. The external file must be connected with sequential access to a unit. (See "OPEN Statement" on page 134.)

Syntax

```
REWIND un

REWIND ( [UNIT=]un [, ERR=err] [, IOSTAT=ios] )
```

UNIT=un
is the reference to the number of an I/O unit. un can optionally be preceded by UNIT= if the second form of the statement is used. It can be an integer or real arithmetic expression. Its value (after conversion to integer of length 4, if necessary) must be zero or positive; otherwise, an error is detected.

ERR=err
is optional. err is a statement number. If an error occurs in the execution of the REWIND statement, control is transferred to the statement labeled err. That statement must be executable and must be in the same program unit as the REWIND statement. If ERR=err is omitted, execution halts when an error is detected.

IOSTAT=ios
is optional. ios is an integer variable or an integer array element of length 4. Its value is set positive if an error is detected; it is set to zero if no error is detected. VSAM return and reason codes are placed in ios.

If UNIT= is specified, all the parameters can appear in any order; otherwise un must appear first.

If the unit specified by un is connected, it must be connected for sequential access. If it is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

The external file connected to the unit specified by un may or may not exist when the statement is executed. If the external file does not exist, the REWIND statement has no effect. If the external file does exist, an end-of-file is created, if necessary, and the file is positioned at the beginning of the first record.

The REWIND statement causes a subsequent READ or WRITE statement referring to un to read data from or write data into the first record of the external file associated with un.

IBM EXTENSION

The REWIND statement may be used with asynchronous READ and WRITE statements provided that any input/output operation on the file has been completed by the execution of a WAIT statement. A WAIT statement is not required to complete the REWIND operation.

END OF IBM EXTENSION

Transfer is made to the statement specified by the ERR parameter if an error is detected. If the IOSTAT=ios is specified, a positive integer value is assigned to ios when an error is detected. Then execution continues with the statement specified with the ERR parameter, if present, or with the next statement if ERR is not specified. If the ERR parameter and the IOSTAT

parameter are both omitted, program execution is terminated when an error is detected.

Valid REWIND Statements:

```
REWIND (5)

REWIND (3*IN-2,ERR=99999)

REWIND (UNIT=2*IN+2)

REWIND (IOSTAT=IOS,ERR=99999,UNIT=2*IN-10)
```

SAVE STATEMENT

The SAVE statement retains the definition status of the name of a named common block, variable, or array after the execution of a RETURN or END statement in a subprogram.

Because VS FORTRAN saves these names without user action, the SAVE statement serves only as a documentation aid.

Syntax

```
SAVE [name1 [, name2 ] ... ]
```

name
is a named common block name preceded and followed by a slash, a variable name, or an array name. Redundant appearances of an item are not permitted.

Dummy argument names, procedure names, and names of entities in a common block must not appear in a SAVE statement.

A SAVE statement without a list is treated as though it contained the names of all allowable items in that program unit.

The appearance of a named common block in a SAVE statement has the effect of specifying all entities in that named common block.

The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

- Entities specified by SAVE statements.
- Entities in blank common.
- Initially defined entities that have neither been redefined nor become undefined.
- Entities in named common blocks that appear in the subprogram and appear in at least one other program unit that is referring, either directly or indirectly, to that subprogram. The entities in a named common block may become undefined by execution of a RETURN or END statement in another program unit.

Within a function or subroutine subprogram, an entity (that is, a common block, variable, or array) specified by a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram.

If a local entity that is specified by a SAVE statement and is not in a common block is in a defined state at the time a RETURN or END statement is executed in a subprogram, that entity is defined with the same value at the next reference of that subprogram. An entity in a common block never becomes undefined as a result of the execution of a RETURN or END statement in a program unit that does not reference that common block. The entities in a named common block may become undefined or redefined by some other program unit.

STATEMENT FUNCTION STATEMENT

A statement function definition specifies operations to be performed whenever that statement function name appears as a function reference in another statement in the same program unit.

Syntax

```
name ( [ arg1 [, arg2 ] ... ] ) = m
```

name
is the statement function name (see "Names" on page 8).

arg
is a statement function dummy argument. It must be a distinct variable, that is, it may appear only once within the list of arguments. Parentheses must be specified even if no dummy argument is specified.

m
is any arithmetic, logical, or character expression. Any statement function appearing in this expression must have been defined previously. In a function or subroutine subprogram, this expression can contain dummy arguments that appear in the FUNCTION, SUBROUTINE, or ENTRY statements of the same program unit. (See "VS FORTRAN Expressions" on page 25 for evaluation and restrictions of this expression.)

All statement function definitions to be used in a program must follow the specification statements and precede the first executable statement of the program.

The length of a character statement function must be an expression containing only integer constants or names of integer constants.

The expression to the right of the equal sign defines the operations to be performed when a reference to this function appears in a statement elsewhere in the program unit. The expression defining the function must not contain (directly or indirectly) a reference to the function it is defining or a reference to any of the entry point names (PROGRAM, FUNCTION, SUBROUTINE, ENTRY) of the program unit where it is defined.

If the expression is an arithmetic expression, its type may be different from the type of the name of the function. Conversions are made as described for the assignment statement.

The dummy arguments enclosed in parentheses following the function name are dummy variables for which the arguments given in the function reference are substituted when the function reference is encountered. The same dummy arguments may be used in more than one statement function definition, and may be used as variables of the same type outside the statement function definitions, including dummy arguments of subprograms. The length specification of a dummy argument of type character must be an arithmetic expression containing only integer constants or names of integer constants.

An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument. It cannot be a character expression involving concatenation of one or more operands whose length specification is an asterisk.

If an actual argument is of type character, the associated dummy argument must be of type character and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument is greater than the length of an associated dummy argument, the leftmost characters of the actual argument are associated with the dummy argument.

The name of a statement function must not appear in an EXTERNAL statement and must not be used as an actual argument.

For example, The statement:

```
FUNC(A,B) = 3.*A+B**2.+X+Y+Z
```

defines the statement function FUNC, where FUNC is the function name and A and B are the dummy arguments. The expression to the right of the equal sign defines the operations to be performed when the function reference appears in an arithmetic statement.

The function reference might appear in a statement as follows:

```
C = FUNC(D,E)
```

This is equivalent to:

```
C = 3.*D+E**2.+X+Y+Z
```

Notice the correspondence between the dummy arguments A and B in the function definition and the actual arguments D and E in the function reference.

Valid Statement Function Definitions and References:

Definition	Reference
`SUM(A,B,C,D) = A+B+C+D`	`NET = GROS-SUM(TAX,COVER,HOSP, STOC)`
`FUNC(Z) = A+X*Y*Z`	`ANS = FUNC(RESULT)`
`VALID(A,B) = .NOT. A .OR. B`	`VAL = TEST .OR. VALID(D,E)`
	`BIGSUM = SUM(A,B,SUM(C,D,E,F),G(I))`

Invalid Statement Function Definitions:

`SUBPRG(3,J,K)=3*I+J**3`	Arguments must be variables.
`SOMEF(A(I),B)=A(I)/B+3.`	Arguments must not be array elements.
`SUBPROGRAM(A,B)=A**2+B**2`	Function name exceeds limit of six characters.
`3FUNC(D)=3.14*E`	Function name must begin with an alphabetic character.
`BAD(A,B)=A+B+BAD(C,D)`	Recursive definition not permitted.
`NOGOOD(A,A)=A*A`	Arguments are not distinct variable names.

Invalid Statement Function References:

(The functions are defined as above.)

`WRONG = SUM(TAX,COVER)`	Number of arguments does not agree with above definition.
`MIX = FUNC(I)`	Type of argument does not agree with above definition.

STATEMENT NUMBERS

Statement numbers identify statements in a VS FORTRAN program. Any statement can have a number. A statement can be written in either fixed form or free form. See "Source Language Statements" on page 5.

Fixed Form Statement Numbers

Fixed form statement numbers have the following attributes:

- They contain one to five decimal digits (not zero) and are on a noncontinued line.
- Blanks and leading zeros are ignored.
- They are in columns 1 through 5.

IBM EXTENSION

Free Form Statement Numbers

Free form statement numbers have the following attributes:

- They must be the first nonblank characters (digits) on an initial line.
- Blanks and leading zeros are ignored.
- No blanks are needed between the statement number and the first nonblank character following.

END OF IBM EXTENSION

See "ASSIGN Statement" on page 46 for information on assignment of statement numbers.

STOP STATEMENT

The STOP statement terminates the execution of the object program and may display a message.

Syntax

```
STOP [n]
STOP ['message']
```

n
a string of 1 through 5 decimal digits.

'message'
a character constant enclosed in apostrophes and containing alphameric and/or special characters. Within the literal, an apostrophe is indicated by two successive apostrophes.

If either *n* or *'message'* is specified, STOP displays the requested information. For further information, see VS FORTRAN Application Programming: Guide.

SUBROUTINE STATEMENT

The SUBROUTINE statement identifies a subroutine subprogram.

Syntax

```
SUBROUTINE name [ ( [arg1] [,arg2] ... ] ) ]
```

name
is the subroutine name (see "Names" on page 8).

arg
is a distinct dummy argument (that is, it may appear only once within the statement). There need not be any arguments, in which case the parentheses may be omitted. Each argument used must be a variable or array name, the dummy name of another subroutine or function subprogram, or an asterisk, where the character * denotes a return point specified by a statement number in the calling program.

Because the subroutine is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

The SUBROUTINE statement must be the first statement in the subprogram. The subroutine subprogram may contain any FORTRAN statement except a FUNCTION statement, another SUBROUTINE statement, a BLOCK DATA statement, or a PROGRAM statement. If an IMPLICIT statement is used in a subroutine subprogram, it must follow the SUBROUTINE statement and may only be preceded by another IMPLICIT statement, a PARAMETER, FORMAT, or ENTRY statement.

The subroutine name must not appear in any other statement in the subroutine subprogram. It must not be the same as any name in the program unit or as the PROGRAM name, a subroutine name, or a common block name in any other program unit of the executable program. The subroutine subprogram may use one or more of its arguments to return values to the calling program. An argument so used will appear on the left side of an arithmetic, logical, or character assignment statement, in the list of a READ statement within the subprogram, or as an argument in a CALL statement or function reference that is assigned a value by the subroutine or function referred to.

The dummy arguments (arg1, arg2, arg3,..., argn) may be considered dummy names that are replaced at the time of execution by the actual arguments supplied in the CALL statement.

If a subroutine dummy argument is used as an adjustable array name, the array name and all the variables in the array declarators (except those in common) must be in the dummy argument list. See "Size and Type Declaration of an Array" on page 22.

The subroutine subprogram can be a set of commonly used computations, but it need not return any results to the calling program. For information about using RETURN and END statements in a subroutine subprogram, see "END Statement" on page 77 and "RETURN Statement" on page 164.

Actual Arguments in a Subroutine Subprogram

The actual arguments in a subroutine reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. The use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.

If an actual argument is of type character, the associated dummy argument must be of type character and the length of the actual

argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument is greater than the length of an associated dummy argument, the leftmost characters of the actual argument are associated with the dummy argument.

An actual argument in a subroutine reference must be one of the following:

- An expression, except for a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses (unless the operand is the name of a constant)
- An array name
- An intrinsic function name
- An external procedure name
- A dummy procedure name
- An alternate return specifier (statement number preceded by an asterisk)

An actual argument in a subroutine reference may be a dummy argument name that appears in a dummy argument list within the subprogram containing the reference. An asterisk dummy argument cannot be used as an actual argument in a subprogram reference.

Dummy Arguments in a Subroutine Subprogram

The dummy arguments of a subprogram appear after the subroutine name and are enclosed in parentheses. They are replaced at the time of execution of the CALL statement by the actual arguments supplied in the CALL statement in the calling program.

Dummy arguments must follow certain rules:

- None of the dummy argument names may appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, SAVE, INTRINSIC, or NAMELIST statement except as common block names.
- A dummy argument name must not be the same as the entry point name appearing in a PROGRAM, FUNCTION, SUBROUTINE, ENTRY, or statement function definition in the same program unit.
- The dummy arguments must correspond in number, order, and type to the actual arguments.
- If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, an array element, a substring, or an array. A constant, name of constant, subprogram name, or expression should not be written as an actual argument unless the programmer is certain that the corresponding dummy argument is not assigned a value in the subprogram.
- A referenced subprogram cannot assign new values to dummy arguments that are associated with other dummy arguments within the subprogram or with variables in COMMON.
- The subprogram reserves no storage for the dummy argument, using the corresponding actual argument in the calling program for its calculations. Thus the value of the actual argument changes as soon as the dummy argument changes.

IBM EXTENSION

TRACE OFF STATEMENT

The TRACE OFF statement stops the display of program flow by statement number.

Syntax

```
TRACE OFF
```

TRACE OFF may appear anywhere within a debug packet. After a TRACE ON statement, tracing continues until a TRACE OFF statement is encountered.

TRACE ON STATEMENT

The TRACE ON statement initiates the display of program flow by statement number.

Syntax

```
TRACE ON
```

TRACE ON is executed only when the TRACE option appears in a DEBUG packet. (See "DEBUG Statement" on page 68.) Tracing continues until a TRACE OFF statement is encountered. TRACE ON stays in effect through any level of subprogram CALL or RETURN statement. However, if a TRACE ON statement is in effect and control is given to a program in which the TRACE option is not specified, the statement numbers in that program are not traced.

Each time a statement with an external statement number is executed, a record of the statement number is made on the debug output file.

For a given debug packet, the TRACE ON statement takes effect immediately before the execution of the statement specified in the AT statement.

END OF IBM EXTENSION

UNCONDITIONAL GO TO

See "GO TO Statements" on page 115.

IBM EXTENSION

WAIT STATEMENT

The WAIT statement completes the data transmission begun by the corresponding asynchronous READ or WRITE statement.

Syntax

```
WAIT ( [UNIT=]un, plist ) [list]
```

UNIT=un
un is required. It can optionally be preceded by UNIT=. It is the reference to an I/O unit. un is an unsigned integer expression of length 4.

plist
is a parameter list that contains (in any order) one or more of the following forms:

ID=id
where id is an integer constant or integer expression of length 4. This parameter is required.

If the WAIT is completing an asynchronous READ, the expression id is subject to the following rules:

- No array element in the receiving area of the read may appear in the expression. This also includes indirect references to such elements; that is, reference to or redefinition of any variable or array element associated by COMMON or EQUIVALENCE statement, or argument association with an array element in the receiving area.
- If a function reference appears in the subscript expression of e1 or e2, the function may not be referred to in the expression id. Also, no functions or subroutines may be referred to by the expression that directly or indirectly refers to the subscript function, or to which the subscript function directly or indirectly refers.

COND=i1
where i1 is an integer variable name of length 4. This parameter is optional.

If COND=i1 is specified, the variable i1 is assigned a value of 1 if the input or output operation was completed successfully; 2 if an error condition was encountered; and 3 if an end-of-file condition was encountered while reading. In case of an error or end-of-file condition, the data in the receiving area may be meaningless.

NUM=i2
where i2 is an integer variable name of length 4. This parameter is optional.

If NUM=i2 is specified, the variable i2 is assigned a value representing the number of bytes of data transmitted to the elements specified by the list. If the list requires more data from the record than the record contains, this parameter must be specified. If the WAIT is completing an asynchronous WRITE, i2 remains unaltered.

list
is optional. It is an asynchronous I/O list as specified for the asynchronous READ and WRITE statements.

If a list is included, it must specify the same receiving or transmitting area as the corresponding asynchronous READ or WRITE statement. It must not be specified if the asynchronous READ did not specify a list.

WAIT redefines a receiving area and makes it available for reference, or makes a transmitting area available for redefinition.

The corresponding asynchronous READ or WRITE, which need not appear in the same program unit as the WAIT, is the statement that:

- Was not completed by the execution of another WAIT
- Refers to the same file as the WAIT
- Contains the same value for id in the ID=id form as did the asynchronous READ or WRITE when it was executed

The correspondence between WAIT and an asynchronous READ or WRITE holds for a particular execution of the statements. Different executions may establish different correspondences.

When the WAIT is completing an asynchronous READ, the subscripts in the list may not refer to array elements in the receiving area. If a function reference is used in a subscript, the function reference may not perform I/O on any file.

Valid WAIT Statements:

```
WAIT (8,ID=1) ARRAY(101)...ARRAY(500)

WAIT (9,ID=1,COND=ITEST)

WAIT (8,ID=1,NUM=N)

WAIT (9,ID=2)
```

END OF IBM EXTENSION

WRITE STATEMENTS

The WRITE statements transfer data from storage to an external device or from one internal file to another internal file.

FORMS OF THE WRITE STATEMENT:

IBM EXTENSION

1. WRITE Statement—Asynchronous

END OF IBM EXTENSION

2. WRITE Statement—Formatted with Direct Access
3. WRITE Statement—Formatted with Sequential Access
4. WRITE Statement—Unformatted with Direct Access
5. WRITE Statement—Unformatted with Sequential Access
6. WRITE Statement with Internal Files
7. WRITE Statement with List-Directed I/O

IBM EXTENSION

8. WRITE Statement with NAMELIST

END OF IBM EXTENSION

IBM EXTENSION

WRITE Statement—Asynchronous

The asynchronous WRITE statement transmits data from an array in main storage to an external file.

Syntax

```
WRITE ( [UNIT=]un, ID=id ) list
```

UNIT=un
un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression of length 4. It is the reference to an I/O unit.

ID=id
id is an integer constant or integer expression of length 4. It is the identifier for the WRITE statement.

list
is an asynchronous I/O list that may have any of four forms:

```
e
e1...e2
e1...
...e2
```

where:

e
is the name of an array.

e1 and **e2**
are the names of elements in the same array. The ellipsis (...) is an integral part of the syntax of the list and must appear in the positions indicated.

The unit specified by un must be connected to a file that resides on a sequential or direct access device. The array or array elements specified by e (or e1 and e2) constitute the transmitting area for the data to be written. The extent of the transmitting area is determined as follows:

- If e is specified, the entire array is the transmitting area.
- If e1...e2 is specified, the transmitting area begins at array element e1 and includes every element up to and including e2. The subscript value of e1 must not exceed that of e2.
- If e1... is specified, the transmitting area begins at element e1 and includes every element up to and including the last element of the array.
- If ...e2 is specified, the transmitting area begins at the first element of the array and includes every element up to and including e2.
- If a function reference is used in a subscript of the list, the function reference may not perform I/O on any file.

Execution of an asynchronous WRITE statement initiates writing of the next record on the specified file. The size of the record is equal to the size of the transmitting area. All the data in the area is written.

Given an array with elements of length len, the number of bytes transmitted will be len times the number of elements in the array. Elements are transmitted sequentially from the smallest subscript element to the highest. If the array is multi-dimensional, the leftmost subscript quantity increases most rapidly, and the rightmost least rapidly.

Because the asynchronous WRITE statement can only refer to files with sequential access, REC may not be specified even though the file may be resident on a direct-access device.

There is no FORMAT statement associated with the output data and no conversion takes place.

Any number of program statements may be executed between an asynchronous WRITE and its corresponding WAIT, subject to the following rules:

- No such statement may in any way assign a new value to any array element in the transmitting field. This and the following rules apply also to indirect references to such array elements; that is, assigning a new value to a variable or array elements associated by COMMON or EQUIVALENCE statements, or argument association with an array element in the transmitting area.
- No executable statement may appear that redefines or undefines a variable or array element appearing in the subscript of e1 or e2.
- If a function reference appears in the subscript expression of e1 or e2, the function may not be referred to by any statements executed between the asynchronous WRITE and the corresponding WAIT. Also, no subroutines or function may be referred to that directly or indirectly refer to the subscript function, or to which the subscript function directly or indirectly refers.
- No function or subroutine may be executed that performs input or output on the file being manipulated.

Valid WRITE Statement:

```
WRITE (ID=10, UNIT=2*IN+2) . . . EXPECT(9)
```

END OF IBM EXTENSION

WRITE Statement—Formatted with Direct Access

This statement transfers data from internal storage onto an external device. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The data must be sent to an external file that is connected with direct access to a unit (see "OPEN Statement" on page 134).

Syntax

```
WRITE ( [UNIT=]un, [FMT=]fmt, REC=rec [,ERR=stn]
        [, IOSTAT=ios ] ) [list]
```

UNIT=un

un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression of length 4. It is the reference to an I/O unit.

If UNIT= is not included, un must appear first in the statement. The other parameters may appear in any order.

If UNIT= is included, all the parameters can appear in any order.

FMT=fmt

fmt is a required format identifier. It can optionally be preceded by FMT=.

If FMT= is not included, the format identifier must appear second.

If both UNIT= and FMT= are included, all parameters, except list, can appear in any order.

The format identifier (fmt) can be:

- A statement number
- An integer variable
- A character constant
- A character array element
- A character array name
- A character expression

IBM EXTENSION

- An array name

END OF IBM EXTENSION

The statement number must be the statement number of a FORMAT statement in the same program unit as the WRITE statement.

The integer variable must have been initialized by an ASSIGN statement with the number of a FORMAT statement. The FORMAT statement must be in the same program unit as the WRITE statement.

The character constant must constitute a valid format. The constant must be delimited by apostrophes, must begin with a left parenthesis and end with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. An apostrophe in a constant enclosed in apostrophes is be represented by two consecutive apostrophes.

The character array element must contain character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes

described in the FORMAT statement can be used between the parentheses. Blank characters may precede the left parenthesis and character data may follow the right parenthesis. The length of the format specification must not exceed the length of the character array element.

The character array name must contain character data whose leftmost characters constitute a valid format specification. The length of the format specification may exceed the length of the first element of the array; it is considered the concatenation of all the elements of the array in the order given by array element ordering.

IBM EXTENSION

The array name may be of type integer, real, double precision, logical, or complex.

The data must be a valid format identifier as described under character array name above.

END OF IBM EXTENSION

The character expression may contain concatenations of character constants, character array elements and character array names. Its value must be a valid format specification. The operands of the expression must have length specifications that contain only integer constants or names of integer constants.

REC=rec
rec is an integer expression. It represents the relative position of a record within the file associated with un. Its value after conversion to integer, if necessary, must be greater than zero. The internal record number of the first record is 1. The INQUIRE statement can be used to determine the record number.

If list is omitted, a blank record is transmitted to the output device unless the FORMAT statement referred to contains, as its first specification, a character constant or slashes. In this case the record (or records) indicated by these specifications are transmitted to the output device.

ERR=stn
stn is the number of a statement in the same program unit as the WRITE statement. Transfer is made to stn if an error is detected.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list and can contain variable names, array element names, character substring names, array names, implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 74.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (un,fmt,REC=rec) list
WRITE (un,FMT=fmt,REC=rec) list
WRITE (FMT=fmt,REC=rec,UNIT=un) list
WRITE (REC=1, UNIT=11, FMT='(I9)')
WRITE (0,'(A8)', REC=3)
```

Invalid WRITE Statements:

`WRITE (fmt,un) list`	un must appear before fmt. REC= is required for direct access.
`WRITE (FMT=fmt, un) list`	un must appear first because UNIT= is not included. REC= is required for direct access.
`WRITE (fmt, UNIT=un) list`	FMT must be used because UNIT= is included. REC= is required for direct access.
`WRITE FMT=fmt, list`	FMT must not be specified. REC= is required for direct access.

If this WRITE statement is encountered, the unit specified must exist and the file must be connected for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

DATA TRANSMISSION: A WRITE statement with FORMAT starts data transmission at the beginning of a record specified by REC=rec. The format codes in the format specification fmt are taken one by one and associated with every item of the list in the order they are specified. The data is taken from the item of the list, converted according to the corresponding format code, and the number of character data specified by the format code is transmitted onto the record of the external file. Data transmission stops when data has been taken from every item of the list or when the end of the record specified by rec is reached.

If the list is not specified and the format specification starts with an I, E, F, D, G, or L, or is empty (that is, FORMAT( )), the record is filled with blank characters and the relative record number rec is increased by one.

IBM EXTENSION

This is also true when the format specification starts with a G, Q, or Z format code.

END OF IBM EXTENSION

DATA AND I/O LIST: The length of every FORTRAN record is specified in the RECL parameter of the OPEN statement. If the length of the record rec is greater than the total amount of data specified by the format codes used during transmission of data, an error is detected, but as much data as can fit into the record is transmitted. If the format specification indicates (for example, slash format code) that data be transmitted to the next record, then the relative record number rec is increased by one and data transmission continues.

After successful execution of the WRITE statement, the value of the NEXTREC variable specified in the INQUIRE statement is set to the relative record number of the last record written,

incremented by one. If an error is detected, the NEXTREC variable contains the relative record number of the record being written.

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be written, only that an error occurred during transmission of data. If IOSTAT=ios is specified, a positive integer value is assigned to stn when an error is detected. Then execution continues with the statement specified with ERR, if present, or with the next statement, if ERR is not specified. If ERR and IOSTAT are both omitted, program execution is terminated when an error is detected.

WRITE Statement—Formatted with Sequential Access

This statement transfers data from internal storage onto an external I/O device. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The data must be sent to an external file that is connected with sequential access to a unit (see "OPEN Statement" on page 134).

Syntax

```
WRITE ( [UNIT=]un, [FMT=]fmt [, ERR=stn] [, IOSTAT=ios] )
        [list]
PRINT fmt [, list]
```

UNIT=un

un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression or an asterisk (*). It is the reference to an I/O unit.

If UNIT= is included, FMT= must be used and all the parameters can appear in any order.

If UNIT= is not included, un must appear first in the statement. The other parameters may appear in any order.

In the form of the PRINT statement where un is not specified, or in the form of a WRITE statement where un is an asterisk, un is installation dependent.

FMT=fmt

fmt is a required format identifier. It can optionally be preceded by FMT=.

If FMT is not included, the format identifier must appear second.

If both UNIT= and FMT= are included, all parameters, except list, can appear in any order.

The format identifier (fmt) can be:

A statement number
An integer variable
A character constant
A character array element
A character array name
A character expression

IBM EXTENSION

An array name

END OF IBM EXTENSION

See "WRITE Statement—Formatted with Direct Access" on page 181 for explanations of these format identifiers.

ERR=stn

stn is the number of a statement in the same program unit as the WRITE statement. Transfer is made to stn if an error is detected.

IOSTAT=ios

ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list. It can contain variable names, array elements, character substring names, array names, implied DO lists, and expressions. In the PRINT statement, if the list is not present, the comma must be omitted. See "Implied DO in an Input/Output Statement" on page 74.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE and PRINT Statements:

```
WRITE (un,fmt) list

WRITE (un, FMT=fmt) list

WRITE (*,fmt) list

WRITE (UNIT=un, FMT=fmt) list      FMT=fmt can appear first.

WRITE(IOSTAT=IOS,ERR=99999,FMT=*,UNIT=2*IN+3)

WRITE(IN+8,NAMEOT,IOSTAT=IACT(1),ERR=99999)

PRINT *, list

PRINT fmt, list

PRINT fmt
```

Invalid WRITE and PRINT Statements:

`WRITE (fmt,un)`	*un* must appear first before *fmt*.
`WRITE (FMT=fmt,un) list`	*un* must appear first because UNIT= is not included.
`WRITE (fmt,UNIT=un) list`	FMT must be used because UNIT= is included.
`PRINT FMT=fmt, list`	FMT must not be used with PRINT.

If the unit specified by *un* is connected, it must be connected for sequential access. If it is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

DATA TRANSMISSION: A WRITE statement with FORMAT starts data transmission at the beginning of a record. The format codes in the format specification *fmt* are taken one by one and associated with every item of the list in the order they are specified. The data is taken from the item of the list, converted according to the corresponding format code and the number of character data specified by the format code is transmitted onto the record of the external file. Data transmission stops when data has been taken from every item of the list.

If the list is not specified and the format specification starts with an I, E, F, D, G, or L, or is empty (that is, FORMAT( )), a blank record is written out.

IBM EXTENSION

This is also true when the format specification starts with a Q or Z format code.

The WRITE statement can be used to write over an end of file and extend the external file. An ENDFILE, BACKSPACE, CLOSE, or REWIND statement will then reinstate the end of file.

END OF IBM EXTENSION

After execution of a sequential WRITE or PRINT, no record exists in the file following the last record transferred by that statement.

DATA AND I/O LIST: The amount of character data specified by all the format codes used during the transmission of the data defines the length of the FORTRAN record (also called a logical record). A single WRITE statement may create several FORTRAN records. This occurs when a slash format code is encountered in the format specification or when the I/O list exceeds the format specification which causes the FORMAT statement to be used in full or part again. (See "FORMAT Statement" on page 90.)

The VS FORTRAN Application Programming: Guide describes how to associate FORTRAN records (that is, logical records) and physical records on an external I/O device.

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be written, only that an error occurred during transmission of data. If IOSTAT=ios is specified, a positive integer value is assigned to stn when an error is detected. Then execution continues with the statement specified with ERR, if present, or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, execution is terminated when an error is detected.

WRITE Statement—Unformatted with Direct Access

The statement transfers data without conversion from internal storage onto an external I/O device. The data must be sent to an external file that is connected with direct access to a unit (see "OPEN Statement" on page 134).

Syntax

```
WRITE ( [UNIT=]un, REC=rec [, ERR=stn] [, IOSTAT=ios] )
        [list]
```

UNIT=un
un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression of length 4. It is the reference to an I/O unit.

If UNIT= is not included, un must appear first in the statement. The other parameters may appear in any order.

If UNIT= is included, all the parameters may appear in any order.

REC=rec
rec is a relative record number. It is an integer expression that must be greater than zero. It represents the relative position of a record within the external file associated with un. The relative record number of the first record is 1.

ERR=stn
stn is the number of a statement in the same program unit as the WRITE statement. Transfer is made to stn if an error is detected.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list and can contain variable names, array elements, character substring names, array names, implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 74.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (un,REC=rec) list

WRITE (REC=rec,UNIT=un) list

WRITE (IOSTAT=IOS, ERR=99999, REC=IN-3, UNIT=IN+6)

WRITE (IOSTAT=IACT(1), REC=2*IN-7, UNIT=2*IN+1) EXPECT(3)

WRITE (REC=1, UNIT=11) EXPECT(1)
```

Invalid WRITE Statements:

```
WRITE (REC=rec,un) list        UNIT must be used.

WRITE (un) list                REC=rec must be specified.
```

If the unit specified by un is encountered, it must exist and the file must be connected for direct access. If the unit is not connected to a file, it is assumed to have been preconnected

through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

DATA TRANSMISSION: A WRITE statement without conversion starts data transmission at the record specified by rec. The data is taken from the items of the list in the order in which they are specified and transmitted onto the record rec of the external file. Data transmission stops when data has been transferred from every item of the list.

DATA AND I/O LIST: The length of every FORTRAN record is specified in the RECL parameter of the OPEN statement. If the length of the record rec is greater than the total amount of data transmitted from the items of the list, the remainder of the record is filled with zeros. If the length of the record rec is smaller than the total amount of data transmitted from the items of the list, as much data as can fit in the record is written, the internal record number is increased by one. The INQUIRE statement can be used to determine the record number.

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be written, only that an error occurred during transmission of data. If IOSTAT=ios is specified, a positive integer value is assigned to ios when an error is detected. Then execution continues with the statement specified with ERR, if present, or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, execution is terminated when an error is detected.

WRITE Statement—Unformatted with Sequential Access

This statement transfers data without conversion from internal storage onto an external I/O device. The data must be sent to an external file that is connected with sequential access to a unit (see "OPEN Statement" on page 134).

Syntax

```
WRITE ( [UNIT=]un [, ERR=stn] [, IOSTAT=ios] ) [list]
```

UNIT=un
un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression of length 4. It is the reference to an I/O unit.

If UNIT= is not included, un must appear first in the statement. The other parameters may appear in any order.

If UNIT= is included, all the parameters may appear in any order.

ERR=stn
stn is the number of a statement in the same program unit as the WRITE statement. Transfer is made to stn if an error is detected.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list and can contain variable names, array elements, character substring names, array names, implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 74.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (un) list

WRITE (UNIT=un) list

WRITE(5) EXPECT(4)
```

Invalid WRITE Statement:

```
WRITE un,list                un must be in parentheses.
```

DATA TRANSMISSION: A WRITE statement without conversion starts data transmission at the beginning of a record. The data is taken from the items of the list in the order in which they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item of the list.

After execution of a sequential WRITE statement, no record exists in the file following the last record transferred by that statement.

IBM EXTENSION

The WRITE statement writes over an end of file and extends the external file. An END FILE, BACKSPACE, CLOSE, or REWIND statement will then reinstate the end of file.

END OF IBM EXTENSION

DATA AND I/O LIST: The amount of character data specified by the items of the list defines the length of the FORTRAN record (also called a logical record). A single WRITE statement creates only one FORTRAN record.

The VS FORTRAN Application Programming: Guide describes how to associate FORTRAN records (that is, logical records) and physical records on an external I/O device.

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be written, only that an error occurred during transmission of data. If IOSTAT=ios is specified, a positive integer value is assigned to ios when an error is detected. Then execution continues with the statement specified with ERR, if present, or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, execution is terminated when an error is detected.

WRITE Statement with Internal Files

This statement transfers data from one or more areas in internal storage to another area in internal storage. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The receiving area in internal storage is called an internal file.

Syntax

```
WRITE ( [UNIT=]un, [FMT=]fmt [, ERR=stn] [, IOSTAT=ios] )
        [list]
```

UNIT=un
un is the reference to an area of storage called an internal file. It can optionally be preceded by UNIT=. It can be the name of a character variable, character array, character array element, or character substring.

If UNIT= is included, FMT= must be used. If UNIT= is not included, the unit reference must appear first.

FMT=fmt
is the format specification. It may optionally be preceded by FMT=.

If FMT= is not included, the format specification must appear second.

If both UNIT= and FMT= are included, all parameters, except list, may appear in any order.

The format specification can be:

- A statement number
- An integer variable
- A character constant
- A character array element
- A character expression

IBM EXTENSION

- An array name

END OF IBM EXTENSION

See "WRITE Statement—Formatted with Direct Access" on page 181 for explanations of these format specifications.

ERR=stn
stn is the number of a statement in the same program unit as the WRITE statement. Transfer is made to stn if an error is detected.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list and can contain variable names, array elements, character substring names, array names, implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 74.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Neither the format specification (fmt) nor an item in the list (list) can be:

- Contained in the area represented by un
- Associated with any part of un through EQUIVALENCE, COMMON, or argument passing

Valid WRITE Statements:

```
WRITE (un,fmt) list

WRITE (un,FMT=fmt) list

WRITE (FMT=fmt,UNIT=un) list

WRITE (IOSTAT=IOS, ERR=99999, FMT='(A9)', UNIT=CHAR(1:5)) '1 2 3'

WRITE (CHAR(1:5), '(A9)', IOSTAT=IACT(1)) '4 5 6'
```

Invalid WRITE Statements:

`WRITE (fmt,un) list`	un must appear first before fmt.
`WRITE (FMT=fmt,un) list`	un must appear first because UNIT= is not included.
`WRITE (fmt,UNIT=un) list`	FMT must be used because UNIT= is included.

DATA TRANSMISSION: A WRITE statement starts data transmission at the beginning of the area specified by un. The format codes in the format specification fmt are taken one by one and associated with every item of the list in the order they are specified. Data is taken from the item of the list, converted according to the format code, and the number of character data specified by the format code is moved into the storage area un. Data transmission stops when data has been moved from every item of the list.

If un is a character variable, a character array element, or a character substring name, it is treated as one record only in relation to the format specification.

If un is a character array name, each array element is treated as one record in relation to the format specification.

DATA AND I/O LIST: The length of a record is the length of the character variable, character substring name, or character array element specified by un when the WRITE statement is executed.

If the length of the record is greater than the amount of data specified by the items of the list and the associated format specification, the remainder of the record is filled with blank characters.

If the length of the record is less than the amount of data specified by the items of the list and the associated format specification, as much data as can fit in the record is transmitted and an error is detected.

The format specification may indicate (for example, slash format code) that data be moved to the next record of storage area un. If un specifies a character variable, a character array element, or a character substring name, an error is detected. If un specifies a character array name, data is moved into the next array element unless the last array element has been reached. In this latter case, an error is detected.

Transfer is made to the statement specified by ERR if an error is detected. No indication is given of which record or records could not be written, only that an error occurred during transmission of data. If IOSTAT=ios is specified, a positive integer value is

assigned to ios when an error is detected. Then execution continues with the statement specified with ERR, if present, or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, execution is terminated when an error is detected.

WRITE Statement with List-Directed I/O

This statement transfers data from internal storage onto an external I/O device. The data must be sent to an external file that is connected with sequential access to a unit. (See "OPEN Statement" on page 134.) The type of the items specified in the statement determines the conversion to be performed.

Syntax

```
WRITE ( [UNIT=]un, [FMT=]* [, ERR=stn] [, IOSTAT=ios] )
        [list]
PRINT * [, list]
```

UNIT=un
un is required. It can optionally be preceded by UNIT=. un is an unsigned integer expression or an asterisk (*). It is the reference to an I/O unit.

If UNIT= is not included, un must appear first in the statement. The other parameters may appear in any order.

If UNIT= is included, all the parameters may appear in any order.

In the form of the PRINT statement where un is not specified or in the form of a WRITE statement where un is an asterisk, un is installation dependent.

FMT=*
An asterisk (*) specifies that a list-directed WRITE has to be executed. It can optionally be preceded by FMT= if un is specified.

If FMT= is not included, the format identifier must appear second.

If both UNIT= and FMT= are included, all parameters, except list, may appear in any order.

ERR=stn
stn is the number of a statement in the same program unit as the WRITE statement. Transfer is made to stn if an error is detected.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

list
is an I/O list and can contain variable names, array elements, character substring names, array names, implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 74.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (un,*) list

WRITE (un,FMT=*) list

WRITE (FMT=*,UNIT=un) list

WRITE (5,*)

WRITE (FMT=*,UNIT=*) FIFTY5,ISEG

WRITE (IOSTAT=IOS, ERR=99999, FMT=*, UNIT=2*IN+3)
      ''''//EXPECT(1)//''''

PRINT *, list
```

Invalid WRITE Statements:

`WRITE (*,un) list`	un must appear before *.
`WRITE (FMT=*,un) list`	un must appear first because UNIT= is not included.
`WRITE(*,UNIT=un) list`	FMT must be used because UNIT= is included.
`PRINT FMT=*, list`	FMT must not be used.

If the unit specified by un is encountered, it must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

DATA TRANSMISSION: A WRITE or PRINT statement with list-directed I/O accessing an external file starts data transmission at the beginning of a record. The data is taken from each item in the list in the order they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item in the list.

After execution of a sequential WRITE or PRINT statement, no record exists in the file following the last record transferred by that statement.

The WRITE or PRINT statement can write over an end of file and extend the external file. An ENDFILE, CLOSE, or REWIND statement will reinstate the end of file.

An external file with sequential access written with list-directed I/O is suitable for printing, because a blank character is always inserted at the beginning of each record as a carrier control character.

DATA AND I/O LIST: The amount of character data specified by the items in the list and the necessary data separators define the length of the FORTRAN record (also called a logical record). A single WRITE or PRINT statement creates only one FORTRAN record.

The VS FORTRAN Application Programming: Guide describes how to associate FORTRAN records (that is, logical records) and physical records on an external I/O device. In particular, a logical record may span over many physical records. A character constant or a complex constant can be split over the next physical record if there is not enough space on the current physical record to contain it all.

Character constants produced:

- Are not delimited by apostrophes
- Are not preceded or followed by a value separator

- Have each internal apostrophe represented externally by one apostrophe
- Have a blank character inserted by the processor for carrier control at the beginning of any record that begins with the continuation of a character constant from the preceding record

Transfer is made to the statement specified by ERR if an error occurs. No indication is given of which record or records could not be written, only that an error occurred during transmission of data. If IOSTAT=stn is specified, a positive integer value is assigned to stn when an error is detected. Then execution continues with the statement specified with ERR, if present, or with the next statement if ERR is not specified. If ERR and IOSTAT are both omitted, execution is terminated when an error occurs.

IBM EXTENSION

WRITE Statement with NAMELIST

This statement transfers data from internal storage onto an external I/O device. The type of the items specified in the NAMELIST statement determines the conversions to be performed.

Syntax

```
WRITE ( un, name [, ERR=stn] [, IOSTAT=ios] )
```

un
un is required. It is an unsigned integer expression of length 4. It is the reference to an I/O unit.

name
is a NAMELIST name. This name must appear as the second parameter in the list and must be the same as the name in a NAMELIST statement that precedes the WRITE statement (see "NAMELIST Statement" on page 132).

ERR=stn
stn is the number of a statement in the same program unit as the WRITE statement. Transfer is made to stn if an error is detected.

IOSTAT=ios
ios is an integer variable or an integer array element. Its value is positive if an error is detected; negative if an end of file is encountered; and zero if no error condition is detected. VSAM return and reason codes are placed in ios.

Valid WRITE Statements:

```
WRITE (un, name)

WRITE (IN+8, NAMEOUT, IOSTAT=IACT(1), ERR=99999)
```

Invalid WRITE Statements:

```
WRITE (name,un)          un must appear before name.

WRITE (un,name) list     list must not be specified.
```

If the unit specified by un is encountered, it must exist and must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

A BACKSPACE or REWIND statement should not be used for a file that is written using NAMELIST. If it is, the results are unpredictable (see "BACKSPACE Statement" on page 54).

DATA TRANSMISSION: A WRITE statement with NAMELIST starts data transmission from the beginning of a record. The data is taken from each item in the NAMELIST with name in the order in which they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item in the NAMELIST name.

After execution of a WRITE statement with NAMELIST, no record exists in the file following the end of the NAMELIST just transmitted.

DATA AND NAMELIST: The NAMELIST name name must appear on the external file.

The number of characters specified by the items in the NAMELIST name and the necessary data separators and identifiers are placed on the external file.

Transfer is made to the statement specified by ERR if an input error occurs. No indication is given of which record or records could not be written, only that an error occurred during transmission of data. If ERR is omitted, execution is terminated when an error occurs.

END OF IBM EXTENSION

APPENDIX A. SOURCE LANGUAGE FLAGGER

The VS FORTRAN compiler can flag FORTRAN statements that do not conform to the syntax of the Full or Subset ANS FORTRAN 1978 Standard. See the ANS manual for subset language flags.

ITEMS FLAGGED FOR FULL ANS LANGUAGE

- FREE option.

  The FIPS option cannot be specified with free-form source. The FIPS flagging is ignored.

- LANGLVL(66) option.

  The FIPS option cannot be specified for the 1966 ANS FORTRAN language. The FIPS flagging is ignored.

GLOBAL ITEMS FLAGGED

- Columns 1 to 5 of a continuation card are not blank.
- The currency symbol ($) is used in a name.
- A name has a redundant, nonconflicting declaration.
- A noncharacter variable has an actual length specified.
- One of the following names is used as an intrinsic function name:

  ALGAMA, ARCOS, ARSIN, CCOS, CDABS, CDCOS, CDEXP, CDLOG, CDSIN, CDSQRT, COTAN, CQABS, CQCOS, CQEXP, CQLOG, CQSIN, CQSQRT, DARCOS, DARSIN, DBLEQ, DCMPLX, DCONJG, DCOTAN, DERFC, DERF, DFLOAT, DGAMMA, DIMAG, DLGAMA, DREAL, ERF, ERFC, GAMMA, HFIX, IMAG, IQINT, LGAMMA, QABS, QARCOS, QARSIN, QATAN, QATAN2, QCMPLX, QCONJG, QCOSH, QCOS, QCOTAN, QDIM, QERFC, QERF, QEXP, QEXTD, QEXT, QFLOAT, QIMAG, QINT, QLOG, QLOG10, QMAX1, QMIN1, QMOD, QREAL, QSIGN, QSINH, QSIN, QSQRT, QTANH, QTAN, SNGLQ.

- Explicit type specification statements for REAL*16; explicit type specification statements for COMPLEX*16 and COMPLEX*32.
- nH in other than a FORMAT statement.

STATEMENTS FLAGGED

- Invalid statement
  - Asynchronous READ statement
  - Asynchronous WRITE statement
  - AT statement
  - DEBUG statement
  - DISPLAY statement
  - EJECT statement
  - INCLUDE statement
  - NAMELIST statement
  - READ statement with NAMELIST

- TRACE statement
- WAIT statement
- WRITE statement with NAMELIST

- CALL statement

  The ampersand (&) character is used in front of a statement number.

- DATA statement

  The statement appears before the end of the specification statements.

  A, Q, Z, or nH constant is used.

  Character constants must correspond to character variables.

- ENTRY statement

  An argument is embedded between slashes.

- EQUIVALENCE statement

  One subscript is specified for a multidimensional array.

- EXTERNAL statement

  A name is preceded by an ampersand (&) character.

- FORMAT statement

  The Q or Z format codes are used.

- FUNCTION statement

  An argument is embedded between slashes.

  A length is specified for a real, logical, integer, or complex function.

- IMPLICIT statement

  A length is specified for a real, logical, integer, or complex range.

  The currency symbol ($) is used as an alphabetic character.

- INTEGER, REAL, COMPLEX, LOGICAL type statements

  Data initialization is specified.

- OPEN statement

  RECL is used with SEQUENTIAL.

- PARAMETER statement

  This statement is preceded by an executable statement, a DATA statement, or a statement function definition.

- SUBROUTINE statement

  An argument is embedded between slashes.

EXECUTION-TIME CAUTIONS

The following items are not flagged. However, they are items that are open to misinterpretation and may cause confusion.

- Array declarators in DIMENSION, INTEGER, REAL, COMPLEX, DOUBLE PRECISION, CHARACTER, and COMMON statements.

  The value of the lower dimension can exceed the value of the upper dimension when it is an expression.

- ASSIGN statement

  A variable containing a statement number can be used as containing an integer value with unpredictable results.

- Assigned GOTO statement

  The index variable may not contain a statement number which is specified in the list of statement numbers.

- Assignment statement

  A character assignment can be made with unpredictable results into a string which is also used on the right-hand side of the equal sign.

- COMMON statement

  The same COMMON block can contain character variables corresponding to noncharacter variables across subroutines.

  The length of the same COMMON block may not be the same across subroutines.

  The same COMMON block may be initialized in more than one BLOCK DATA.

- DO statement

  The value of the m3 expression can be zero.

  Transfer into an inactive DO loop with unpredictable results.

- ENDFILE statement

  Multifiles can be written.

- FUNCTION, SUBROUTINE, ENTRY statements

  The subroutines must be available.

  The subroutines can be called recursively with unpredictable results.

  The number, type, and length of the actual and dummy arguments may not match.

  More than one subroutine may have the same name.

- IMPLICIT statement

  The same letter is redefined with different type or length.

- OPEN statement

  The file is repositioned at the beginning.

- READ statement on an internal file

  Read records until the end of an array even if the file is one record.

- READ statement with FORMAT

Data can be read into the nH field of a FORMAT statement.

- Subscript

  Subscript value may be outside the dimension bounds.

- WRITE statement without format on a DIRECT file.

  Spanned records can be written.

APPENDIX B. FORTRAN-SUPPLIED PROCEDURES

The procedures supplied by VS FORTRAN are called intrinsic functions.

MATHEMATICAL AND CHARACTER FUNCTIONS

These routines provide intrinsic functions for mathematical and character operations. When a VS FORTRAN program requests an intrinsic function, the routine is handled as a called subroutine during link-editing and is either:

- Inserted into the program (inline).
- Included in the load module.

The generic name can be used for a function; VS FORTRAN will select the particular function named, depending upon the precision of the data.

Alternatively, the name of the specific alternative entry point can be used. A prefix to the generic name specifies the alternative entry point and indicates the data type of the result, as shown in Figure 21.

Prefix	Result Data Type
A	REAL (included only for compatibility)
D	REAL *8
Q	REAL *16
C	COMPLEX *8
CD	COMPLEX *16
CQ	COMPLEX *32

Figure 21. Function Routine Prefix Meanings

VS FORTRAN includes mathematical and character subroutines in several categories:

1. Logarithmic and exponential routines
2. Trigonometric foutines
3. Hyperbolic Function routines
4. Miscellaneous Mathematical routines
5. Internal Data Conversion routines
6. Character Manipulation routines

LOGARITHMIC AND EXPONENTIAL ROUTINES

- EXP—Obtain an exponent.

  Alternative entry points: CDEXP, CEXP, CQEXP, DEXP, EXP, QEXP.

- LOG—Obtain a natural logarithm.

  Alternative entry points: ALOG, CDLOG, CLOG, CQLOG, DLOG, LOG, QLOG.

- LOG10—Obtain a common logarithm.

  Alternative entry points: ALOG10, DLOG10, LOG10, QLOG10.

- SQRT—Obtain a square root.

  Alternative entry points: CDSQRT, CQSQRT, CSQRT, DSQRT, QSQRT, SQRT.

TRIGONOMETRIC ROUTINES

- ACOS—Obtain an arccosine.

  Alternative entry points: ACOS, ARCOS, DACOS, DARCOS, QARCOS.

- ASIN—Obtain an arcsine.

  Alternative entry points: ARSIN, ASIN, DARSIN, DASIN, QARSIN.

- ATAN/ATAN2—Obtain an arctangent.

  Alternative entry points: ATAN, ATAN2, DATAN, DATAN2, QATAN, QATAN2.

- COS—Obtain a cosine.

  Alternative entry points: CCOS, CDCOS, COS, CQCOS, DCOS, QCOS.

- COTAN—Obtain a cotangent.

  Alternative entry points: COTAN, DCOTAN, QCOTAN.

- SIN—Obtain a sine.

  Alternative entry points: CDSIN, CQSIN, CSIN, DSIN, QSIN, SIN.

- TAN—Obtain a tangent.

  Alternative entry points: DTAN, QTAN, TAN.

HYPERBOLIC FUNCTION ROUTINES

- COSH—Obtain a hyperbolic cosine.

  Alternative entry points: COSH, DCOSH, QCOSH.

- SINH—Obtain a hyperbolic sine.

  Alternative entry points: DSINH, QSINH, SINH.

- TANH—Obtain a hyperbolic tangent.

  Alternative entry points: DTANH, QTANH, TANH.

MISCELLANEOUS MATHEMATICAL ROUTINES

- ABS—Obtain an absolute value.

  Alternative entry points: ABS, CABS, CDABS, CQABS, DABS, IABS, QABS.

- AINT—Truncation of a real number.

  Alternative entry points: AINT, DINT, QINT.

- ANINT—Obtain nearest whole number.

  Alternative entry points: ANINT, DNINT.

- CONJG—Obtain conjugate of a complex argument.

  Alternative entry points: CONJG, DCONJG, QCONJG.

- DIM—Obtain a positive difference.

  Alternative entry points: DIM, DDIM, IDIM, QDIM.

- DPROD—Obtain a double precision product.

- ERF—Error function for normal curve.

  Alternative Entry points: DERF, ERF, QERF.

- ERFC—Error function complement for normal curve.

  Alternative Entry points: DERFC, ERFC, QERFC.

- GAMMA—Gamma function.

  Alternative Entry points: DGAMMA, GAMMA.

- IMAG—Obtain imaginary part of complex argument.

  Alternative Entry points: AIMAG, DIMAG, IMAG, QIMAG.

- LGAMMA—Log-gamma function.

  Alternative Entry points: ALGAMA, DLGAMA, LGAMMA.

- MAX—Obtain the largest value.

  Alternative Entry points: AMAX1, DMAX1, MAX, MAX0, MAX1, QMAX1.

- MIN—Obtain the smallest value.

  Alternative Entry points: AMIN1, AMIN0, DMIN1, MIN, MIN0, MIN1, QMIN1.

- MOD—Obtain a remainder.

  Alternative Entry points: AMOD, DMOD, MOD, QMOD.

- NINT—Obtain nearest integer.

  Alternative Entry points: IDNINT, NINT.

- SIGN—Transfer of sign.

  Alternative Entry points: DSIGN, ISIGN, QSIGN, SIGN.

CHARACTER MANIPULATION ROUTINES

- CHAR—Return the character corresponding to the position in the collating sequence of the input argument.
- ICHAR—Return the position in the collating sequence of the input argument.
- INDEX—Obtain location of character substring.
- LEN—Obtain length of character item.
- LGE—Alphamerically greater than or equal.
- LGT—Alphamerically greater than.
- LLE—Alphamerically less than or equal.
- LLT—Alphamerically less than.

INTERNAL DATA CONVERSION GENERIC FUNCTION DESCRIPTIONS

The following are the generic function names of the internal data conversion routines.

- CMPLX—Convert to complex.

  Alternative entry points: CMPLX, DCMPLX, QCMPLX.

- DBLE—Convert to double precision.

  Alternative entry points: DBLE, DBLEQ, DFLOAT.

- INT—Convert to integer.

  Alternative entry points: HFIX, IDINT, IFIX, INT, IQINT.

- QEXT—Convert to real extended precision.

  Alternative entry points: QEXT, QEXTD, QFLOAT.

- REAL—Convert to real.

  Alternative entry points: DFLOAT, DREAL, FLOAT, QFLOAT, QREAL, REAL, SNGL, SNGLQ.

- SNGL—Convert to single precision.

  Alternative entry points: SNGL, SNGLQ.

APPENDIX C. IBM AND ANS FORTRAN FEATURES

Either the old FORTRAN (LANGLVL(66)) or the current FORTRAN (LANGLVL(77)) compiler option is provided at the time of compilation. The following groups of features are listed in this appendix:

- New ANS FORTRAN 1977 features

  General features
  New statements
  New features in old statements

- Old IBM extensions now in ANS FORTRAN 1977
- IBM extensions not in ANS FORTRAN 1977
- LANGLVL(66) features not in VS FORTRAN

NEW ANS FORTRAN 1977 FEATURES

The following new features of the 1977 American National Stantard (ANS) FORTRAN (not supported by the old IBM OS and DOS FORTRAN compilers) are supported in VS FORTRAN.

GENERAL FEATURES

- May use asterisk comment indicator in column one.
- Comment before continuation is allowed anywhere in the program unit. Blank card is treated as a comment.
- External unit identifier may be an integer expression.
- Direct-access input/output (syntax different from IBM's).
- Storage-to-storage input/output (Internal File).
- Specified ignoring of input blanks.
- Expressions are allowed in output lists.
- Character data type is allowed.
  - May include character substrings.
  - The collating sequence may be altered.
- Subroutines without RETURN.
  - END in subroutine is the same as RETURN.
- Functions (and their entry points) may exist without arguments.
- Dummy argument may be defined if actual argument is in common.
- Array elements are allowed in statement function definitions.
- Array names without subscripts are allowed in the EQUIVALENCE statement.
- Complex data may be defined through real components.
- Variables used in adjustable dimensions and lengths may be redefined without any effect on size of array.
- Integer expressions are allowed in array declarators.

- Nonunity lower bounds for arrays are allowed.
- Nonpositive subscript values are allowed.
- Named BLOCK DATA subroutines are allowed.
- Executable statements that cannot be reached are allowed.
- ANINT, CHAR, DNINT, DPROD, ICHAR, IDNINT, INDEX, LEN, and NINT are recognized as FORTRAN-supplied function names.
- DARCOS and DARSIN functions have different names: DACOS and DASIN.
- Logical operators .EQV. and .NEQV., are allowed.
- A number is permitted on nonexecutable statements.
- Comparison of complex operands with equal and not equal relationals is allowed.
- Exponentiation of complex with complex is allowed.
- All specification statements must precede all DATA statements.
- Negative values for input or output unit identifiers is prohibited.
- Literal format cannot be used for input.
- H format cannot be used for input.
- Use of a slash as a value separator in list-directed input is allowed.
- Character function is allowed.
- Unspecified width is allowed in A format.

NEW STATEMENTS

- Block IF, ELSE IF, ELSE, END IF statements
- CHARACTER type statement
- CLOSE statement
- DOUBLE PRECISION type statement
- INQUIRE statement
- INTRINSIC statement
- OPEN statement
- PARAMETER statement
- PROGRAM statement
- SAVE statement

- BACKSPACE statement:
  - UNIT, ERR, and IOSTAT may be used.
- COMMON statement:
  - Commas are optional.
- DATA statement:
  - Implied DO statement is allowed.
  - Type conversion is allowed.
  - Commas after nonterminal slashes are optional.
- DIMENSION statement:
  - Specification can be negative or zero.
  - Both lower and upper bound can be names of constants or expressions.
- DO statement:
  - Loops may be indexed by nonpositive values.
  - Loops may be indexed by integer, real, or double precision values.
  - Backward loops may be used.
  - Zero trip loops may be used.
  - Control variable is defined on completion.
  - Control variable may be real or double precision.
  - Terminal statements are allowed with computed GO TO, PAUSE, LOGICAL IF, STOP, or RETURN. They are not allowed with block IF.
  - Comma is optional following terminal statement number.
  - Subscript values can be negative or zero.
  - Parameters may be any arithmetic expression except complex.
  - Parameters may be redefined in loop with no effect on loop control.
  - A block IF statement in the DO range must be entirely within the range of the DO.
  - The range of a DO within a block IF must be entirely contained within the block.
  - Transfer may be made into any active loop.
  - DO may be ended by any fall-through statement.
  - Comma may be used before control variable.
- END statement:
  - May be numbered.
  - Implies STOP or RETURN.
  - Is executable.
- ENDFILE statement:

- UNIT, ERR, and IOSTAT may be used.

- EXTERNAL statement:

  An ampersand (&) character as the first character of a name is not permitted for compiler option LANGLVL(77). Any name that appears in an EXTERNAL statement is considered as the name of a user-supplied subroutine.

- FORMAT statement:
  - BN and BZ specify ignoring of input blanks.
  - Unlimited parentheses may be used.
  - The label ASSIGNED may be the number of a FORMAT statement.
  - Field width is optional in A*w*.
  - Explicit *n*P scale factor may be used.
  - Ew.dEe, Gw.dEe, Iw.d, SP, SS, S, TLc, and TRc field descriptors may be used.
  - Colon may be used as scan terminator.
  - Optional commas may be used with slashes and colons.
- GO TO statement, Assigned:
  - List of statement numbers is optional.
  - Comma outside parentheses is optional.
- GO TO statement, Computed:
  - Index may be an integer expression.
  - Comma may be outside parentheses.
- IMPLICIT statement:
  - More than one may be used in a program unit.
  - IMPLICIT may be preceded by ENTRY, FORMAT, or PARAMETER statements and must precede all other specification statements except PARAMETER statements.
  - DOUBLE PRECISION and CHARACTER type statements are included.
- PRINT statement:
  - FORMAT designator may be a character constant.
- READ statements:
  - FORMAT designator may be a character constant.
  - UNIT, ERR, and IOSTAT may be used.
- RETURN statement:
  - Index may be an integer expression.
- REWIND statement:
  - UNIT, ERR, and IOSTAT may be used.
- STOP statement:
  - Quoted literal is allowed.
  - A character constant is permitted.

- Auxiliary input and output statements:
  - UNIT and ERR may be used.
- WRITE statement:
  - May not be used after ENDFILE in sequential input or output.
  - FORMAT designator may be a character constant.
  - UNIT, FMT, REC, and IOSTAT may be used.

OLD IBM EXTENSIONS NOW IN ANS FORTRAN 1977

The following items supported as IBM extensions in old IBM OS and DOS FORTRAN compilers are now part of the 1977 ANS FORTRAN language. These items are also supported in VS FORTRAN.

- Literals are enclosed in apostrophes.
- STOP and PAUSE statements:
  - Decimal digits are supported.
  - STOP statement string is accessible.
  - Quoted literal in PAUSE statement is supported.
- T format is accepted as a field descriptor.
- Computed GO TO index out of range.
- All combinations of arithmetics across equal sign.
- Mixed-mode arithmetic.
- Mixed-mode relationals.
- Successive exponentiations.
- Generalized subscripts.
- Seven-dimensional arrays.
- END in READ.
- ERR in READ and WRITE.
- Short form of READ and PRINT.
- Sequential list-directed input/output.
- Asterisks for undersized output fields.
- IMPLICIT statement.
- Array names in DATA statement.
- ENTRY statement.
- Alternative returns from subroutines.
- Function and entry names in type statements.
- Generic facility.
- Additional processor-supplied functions.

IBM EXTENSIONS NOT IN ANS FORTRAN 1977

The following IBM extensions are supported by old IBM OS and DOS FORTRAN compilers but are not part of the 1977 ANS FORTRAN. They will continue to be supported in VS FORTRAN as IBM extensions.

Some of the following features are available only under the compiler option described in the next section, "LANGLVL(66) Features Not in VS FORTRAN."

- NAMELIST statement.
- Hexadecimal.
- Double Precision Complex.
- Z and Q format descriptor.
- G format for integer and logical.
- ALGAMA, ARCOS, ARSIN, CCOS, CDABS, CDCOS, CDEXP, CDLOG, CDSIN, CDSQRT, COTAN, CQABS, CQCOS, CQEXP, CQLOG, CQSIN, CQSQRT, DARCOS, DARSIN, DBLEQ, DCMPLX, DCONJG, DCOTAN, DERFC, DERF, DFLOAT, DGAMMA, DIMAG, DLGAMA, DREAL, ERF, ERFC, GAMMA, HFIX, IMAG, IQINT, LGAMMA, QABS, QARCOS, QARSIN, QATAN, QATAN2, QCMPLX, QCONJG, QCOSH, QCOS, QCOTAN, QDIM, QERFC, QERF, QEXP, QEXTD, QEXT, QFLOAT, QIMAG, QINT, QLOG, QLOG10, QMAX1, QMIN1, QMOD, QREAL, QSIGN, QSINH, QSIN, QSQRT, QTANH, QTAN, SNGLQ.
- CALL DVCHK, CALL DUMP/PDUMP, CALL EXIT, CALL OVERFL.
- Asynchronous READ, WRITE, and WAIT.
- Extended Precision for REAL and COMPLEX.
- Extended debug facility.
- Hexadecimal constants in Z format are allowed.
- Free form source statements.
- The currency symbol ($) used as alphabetic character.
- Data initialization in type specification statements.
- Optional length specification in specification statements (INTEGER, REAL, COMPLEX, LOGICAL) and in FUNCTION statements.
- Mixed mode expressions involving complex and double precision.
- FORMAT identifier may be an array name (other than character type).
- Continuation line may have anything in columns 1 through 5 other than "C" in column 1.
- RETURN statement is the same as STOP in a main program.
- Partitioned data sets.
- Closing of data set on ABEND.
- STOP*n* is allowed, where *n* equals a return code.

LANGLVL(66) FEATURES NOT IN VS FORTRAN

LANGLVL(66) instructs the compiler to compile a program according to the 1966 FORTRAN language. Listed here are some of the features of LANGLVL(66) that are not in LANGLVL(77). These items are not compatible with VS FORTRAN.

- Character constants may be assigned to integer, real, complex, or logical in a DATA statement.
- The ampersand (&) is included in the character set.
- The ampersand (&) must be used instead of the asterisk (*) for an alternate return.
- A program name can only be specified as a compiler option.
- Arguments are received by value.
- Dummy arguments can be enclosed in slashes.
- DARCOS and DARSIN used as function names are recognized as FORTRAN-supplied functions; DACOS and DASIN are recognized as user-supplied function names.
- DEFINE FILE statement.
- DO statement and implied DO in I/O:

  Loops are always executed at least once.
- EQUIVALENCE statement. (Accept a multidimensional array with one subscript.)
- EXTERNAL statement:

  If a FORTRAN-supplied function name appears in an EXTERNAL statement preceded by an ampersand (&) it is considered a user-supplied function name. If it is not preceded by an ampersand (&), it is considered a FORTRAN-supplied function name except as described below. The following names are always considered user-supplied function names if they appear in an EXTERNAL statement preceded or not by an ampersand (&):

  ABS, AIMAG, AINT, AMAX0, AMAX1, AMIN0, AMIN1, AMOD, CMPLX, CONJG, DABS, DBLE, DBLEQ, DCMPLX, DCONJG, DDIM, DFLOAT, DIM, DIMAG, DINT, DMAX1, DMIN1, DMOD, DREAL, DSIGN, FLOAT, HFIX, IABS, IDIM, IDINT, IFIX, IMAG, INT, IQINT, ISIGN, MAX, MAX0, MAX1, MIN, MIN0, MIN1, MOD, QABS, QCMPLX, QCONJG. QDIM, QEXT, QEXTD, QFLOAT, QIMAG, QINT, QMAX1, QMIN1, QMOD, QREAL, QSIGN, REAL, SIGN, SNGL, SNGLQ.
- FIND statement.
- Function names: ANINT, CHAR, DPROD, DNINT, ICHAR, IDNINT, INDEX, LEN, and NINT are recognized as user-supplied function names.
- GENERIC statement.

  GENERIC means that generic names of FORTRAN-supplied functions will be recognized as generic; if GENERIC is not specified, the automatic function selection facility will not be in effect.
- IBM direct-access READ and WRITE.
- INTRINSIC statement is not recognized as a VS FORTRAN statement.
- PUNCH b, list.

APPENDIX D. EXTENDED ERROR HANDLING SUBROUTINES

IBM provides five subroutines for use in extended error handling: ERRSAV, ERRSET, ERRSTR, and ERRTRA. These subroutines allow access to the option table to alter it dynamically.

Certain option table entries may be protected against alteration when the option table is set up. If a request is made by means of CALL ERRSTR or CALL ERRSET to alter such an entry, the request is ignored. See Figure 24 on page 222 for those IBM-supplied option table entries that cannot be altered.

Changes made dynamically are in effect for the duration of the program that made the change. Only the current copy of the option table in main storage is affected; the copy in the FORTRAN library remains unchanged.

ERRMON SUBROUTINE

The user has the ability to call, from his own program, the FORTRAN error monitor (ERRMON) routine, the same routine used by FORTRAN itself when it detects an error. ERRMON examines the option table for the appropriate error number and its associated entry and takes the actions specified. If a user-exit address has been specified, ERRMON transfers control to the user-written routine indicated by that address. Thus, the user has the option of handling errors in one of two ways: (1) by calling ERRMON without supplying a user-written exit routine; or (2) by calling ERRMON and providing a user-written exit routine.

The error numbers chosen for user subprograms are restricted in range. IBM-designated error conditions have reserved error codes from 000 to 301. Error codes for installation-designated error situations must be assigned in the range 302 to 899. Before you use these subroutines, check with your system administrator about codes and options you can use. The error code is used by FORTRAN to find the proper entry in the option table.

Error Monitor Routines	Figure
Option table preface	Figure 22.
Option table entries	Figure 23.
Option table default values	Figure 24.
Corrective action after error	Figure 25.
Corrective action after mathematical subroutine error	Figure 26.
Corrective action after program interrupt	Figure 27.

To call the ERRMON routine, the following statement is used:

Syntax

```
CALL ERRMON (imes,iretcd,ierno [,data1,data2,...])
```

imes
is the name of an array aligned on a fullword boundary, that contains, in EBCDIC characters, the text of the message. The number of the error condition should be included as part of the text, because the error monitor prints only the text passed to it. The first item of the array contains an integer

whose value is the length of the message. Thus, the first four bytes of the array are not printed. If the message length is greater than the length of the buffer, it is printed on two or more lines of printed output.

iretcd
is an integer variable made available to the error monitor for the setting of a return code. The following codes can be set:

0—The option table or user-exit routine indicates that standard correction is required.

1—The option table indicates that a user exit to a corrective routine has been executed. The function is to be reevaluated using arguments supplied in the parameters data1,data2... . For input/output type errors, the value 1 indicates that standard correction is not wanted.

ierno
is the error condition number in the option table. Should any number not within range of the option table be specified, an error message is printed.

data1,data2...
are variable names in an error-detecting routine for the passing of arguments found to be in error. One variable must be specified for each argument. Upon return to the error-detecting routine, results obtained from corrective action are in these variables. Because the content of the variables can be altered, the locations in which they are placed should be only in the CALL statement to the error monitor; otherwise, the user of the function may have literals or variables destroyed.

Because data1 and data2 are the parameters that the error monitor passes to a user-written routine to correct the detected error, care must be taken to make sure that these parameters agree in type and number in a call to ERRMON and/or in a call to a user-written corrective routine, if one exists.

An example of CALL ERRMON is:

```
CALL ERRMON (MYMSG,ICODE,315,D1,D2)
```

The example states that the message to be printed is contained in an array named MYMSG, the field named ICODE is to contain the return code, the error condition number to be investigated is 315, and arguments to be passed to the user-written routine are contained in fields named D1 and D2.

ERRSAV SUBROUTINE

The CALL ERRSAV statement copies an option table entry into an 8-byte storage area accessible to the FORTRAN programmer. CALL ERRSAV has the format:

Syntax

```
CALL ERRSAV (ierno, tabent)
```

ierno
is the error number in the option table. Should any number not within the range of the option table be used, an error message is printed.

tabent
is the name of an 8-byte storage area in which the option table entry is to be stored.

An example of CALL ERRSAV is:

```
CALL ERRSAV (213,ALTERX)
```

The example states that error number 213 is to be stored in the area named ALTERX.

ERRSET SUBROUTINE

The CALL ERRSET statement permits the user to change up to five different options. It consists of six parameters. The last four parameters are optional, but each omitted parameter must have its place noted by a comma or a zero if succeeding parameters are specified. (Omitted parameters at the end of the list require no place notation.) CALL ERRSET has the format:

Syntax

```
CALL ERRSET (ierno,inoal,inomes,itrace,iusadr,irange)
```

ierno
is the error number in the option table. Should any number not within the range of the option table be used, an error message is printed. (If ierno is specified as 212, there is a special relationship between the ierno and irange parameters. See the explanation of irange.)

inoal
is an integer specifying the number of errors permitted before each execution is terminated. If inoal is specified as either zero or a negative number, the specification is ignored, and the number-of-errors option is not altered. If a value of more than 255 is specified, an unlimited number of errors is permitted.

The value of inoal should be set at 2 or greater if transfer of control to a user-supplied error routine is desired after an error. If this parameter is specified with a value of 1, execution is terminated after only one error.

inomes
is an integer indicating the number of messages to be printed. A negative value specified for inomes suppresses all messages; a specification of zero indicates that the number-of-messages option is not to be altered. If a value greater than 255 is specified, an unlimited number of error messages is permitted.

itrace
is an integer whose value may be 0, 1, or 2. A specification of 0 indicates the option is not to be changed; a specification of 1 requests that no traceback be printed after an error. (If a value other 1 or 2 is specified, the option remains unchanged.)

iusadr
specifies one of the following:

1. The value 1, indicating that the option table is to be set to show no user-exit routine (that is, standard corrective action is to be used when continuing execution).

2. The name of a closed subroutine that is to be executed after the occurrence of the error identified by ierno. The name must appear in an EXTERNAL statement in the source program, and the routine to which control is to be passed must be available at link editing time.

3. The value 0, indicating that the table entry is not to be altered.

irange
serves a double function. It specifies one of the following:

1. An error number higher than that specified in ierno. This number indicates that the options specified for the other parameters are to be applied to the entire range of error conditions encompassed by ierno and irange. (If irange specifies a number lower than ierno, the parameter is ignored, unless ierno specifies the number as 212.)

2. A print control character if ierno specified 212. The value 1 is specified to provide single spacing for an overflow line. If a value other than 1 is specified, no print control is provided.

The default value 0 is assumed if the parameter is omitted (that is, no print control is provided, and the values specified for all parameters apply only to the error condition number in ierno).

EXAMPLES OF CALL ERRSET

Example 1:

```
CALL ERRSET (310,20,5,0,MYERR,320)
```

This example specifies the following:

1. Error condition 310 (ierno).
2. The error condition may occur up to 20 times (inoal).
3. The corresponding error message may be printed up to 5 times (inomes).
4. The default for traceback information is to remain in force (itrace).
5. The user-written routine MYERR is to be executed after each error (iusadr).
6. The same options are to apply to all error conditions from 310 to 320 (irange).

Example 2:

```
CALL ERRSET (212,10,5,2,1,1)
```

This example specifies:

1. Error condition 212.
2. The condition may occur up to 10 times.
3. The corresponding message may be displayed up to 5 times.
4. Traceback information is to be displayed after each error.
5. Standard corrective action is to be executed after an error.
6. Print control is to be employed.

For illustration purposes, this example explicitly specifies all default options except that used in requesting print control.

Example 3:

```
CALL ERRSET (212,0,0,0,0,1)
```

This example illustrates an alternative method of specifying exactly the same options as the second example. It states that no changes are to be made to default settings in requesting print control.

ERRSTR SUBROUTINE

To store an entry in the option table, the following statement is used:

Syntax

```
CALL ERRSTR (ierno,tabent)
```

ierno
is the error number for which the entry is to be stored in the option table. Should any number not within the range of the option table be used, an error is printed.

tabent
is the name of an 8-byte storage area containing the table entry data.

An example of CALL ERRSTR is:

```
CALL ERRSTR (213,ALTREX)
```

The example states that error number 213, stored in ALTREX, is to be restored to the option table.

ERRTRA SUBROUTINE

The CALL ERRTRA statement permits the user to dynamically request a traceback and continued execution. It has the format:

Syntax

```
CALL ERRTRA
```

The CALL ERRTRA statement has no parameters.

Field Contents	Field Length	Default Value	Field Description
Number of entries	4 bytes	152	Number of entries in the option table.
Boundary Alignment	4 bytes	150	Message number of the first table entry.

Figure 22. Option Table Preface

Field Contents	Field Length	Default[1]	Field Description
Number of error occur- rences	1 byte	10[2]	Number of times this error condition should be allowed to occur. When the value of the error count field (below) equals this value, job processing is terminated. Number may range from 0 to to 255. A value of 0 means an unlimited number of occurrences.[3]
Number messages to print	1 byte	5[4]	Number of times the corresponding message is to be printed before message printing is suppressed. A value of 0 means no message is to be printed.
Error count	1 byte	0	The number of times this error has occurred. A value of 0 indicates that no occurrences have been encountered.
Option bits	1 byte	42 (hex)	Eight option bits defined as follows (the default setting is underscored):

Bit	Setting	Explanation:
0	0	No control character supplied for overflow lines.
	1	Control character supplied to provide single spacing for overflow lines.
1	0	Table entry cannot be modified.[5]
	1	Table entry can be modified.
2	0	Fewer than 256 errors hav occurred.
	1	More than 256 errors have occurred. (Add 256 to error count field above to determine the number.
3[6]	0	Do not print buffer with error message.
	1	Print buffer contents.
4	0	Reserved.
5	0	Print messages default number of times only.
	1	Unlimited printing requested; print for every occurrence of error.
6[7]	0	Do not print traceback map.
	1	Print traceback map.
7	0	Reserved.

User exit	4 bytes	1	Indicates where a user corrective routine is available. A value other than 1 specifies the address of the user-written routine.

Figure 23. Option Table Entry

Notes to Figure 23:

1. The default values shown apply to all error numbers (including additional user entries) unless excepted by a footnote.

2. Errors 207, 208, 209, and 215 are set as unlimited, and errors 162, 163, 164, 165, 167, 168, 205, 217, 230, and 240 are set to 1.

3. An unlimited number of errors may cause the FORTRAN job to loop.

4. Errors 162, 163, 164, 165, 167, 168, 230, and 240 are set to 1.

5. The entry for errors 162, 163, 164, 165, 167, 168, 205, 230, and 240 cannot be modified.

6. The entry is set to 0 except for errors 212, 215, 218, 221, 222, 223, 224, 225, 227, 229, and 238.

7. The entry is set to 1 except for error 205.

Error Code	No. of Errors Allowed	No. of Messages Allowed	Print Control	Modifiable Entry	Print Buffer Content	Traceback Allowed	Standard Corrective Action	User Exit
			Option Bits					
150-161	10	5	NA	Yes	No	Yes	Yes	None
162-165	1	1	NA	No	No	Yes	Yes	None
166	10	5	NA	Yes	No	Yes	Yes	None
167	1	1	NA	No	No	Yes	Yes	None
168	1	1	NA	No	No	Yes	Yes	None
169-204	10	5	NA	Yes	No	Yes	Yes	None
205	1	1	NA	No	No	No	No	None
206	10	5	NA	Yes	No	Yes	Yes	None
207	Unlimited	5	NA	Yes	No	Yes	Yes	None
208	Unlimited	5	NA	Yes	No	Yes	Yes	None
209	Unlimited	5	NA	Yes	No	Yes	Yes	None[1]
210	10	5	NA	Yes	No	Yes	Yes[1]	None
211	10	5	NA	Yes	No	Yes	Yes	None
212	10	5	No[2]	Yes	Yes	Yes	Yes	None
213	10	5	NA	Yes	No	Yes	Yes	None
214	10	5	NA	Yes	No	Yes	Yes	None
215	Unlimited	5	NA	Yes	Yes	Yes	Yes	None
216	10	5	NA	Yes	No	Yes	Yes	None
217	1[3]	1	NA	Yes	No	Yes	Yes	None
218	10[4]	4	NA	Yes	Yes[4]	Yes	Yes	None
219	10[5]	5	NA	Yes	No	Yes	Yes	None
220	10	5	NA	Yes	No	Yes	Yes	None
221-225	10	5	NA	Yes	Yes	Yes	Yes	None
226	10	5	NA	Yes	No	Yes	Yes	None
227	10	5	NA	Yes	Yes	Yes	Yes	None
228	10	5	NA	Yes	No	Yes	Yes	None
229	10	5	NA	Yes	Yes	Yes	Yes	None
230	1	1	NA	Yes	No	Yes	No	None
231-237	10	5	NA	Yes	No	Yes	Yes	None
238	10	5	NA	Yes	Yes	Yes	Yes	None
239	10	5	NA	Yes	No	Yes	Yes	None
240	1	1	NA	No	No	Yes	No	None
241-301	10	5	NA	Yes	No	Yes	Yes	None

Figure 24. Option Table Default Values

Notes to Figure 24:

1. No corrective action is taken except to continue execution. For boundary alignment, the corrective action is part of the support for misalign- ment. For error 209, no user corrective action can be specified.

2. If a print control character is not supplied, the overflow line is not shifted to incorporate the print control character. Thus, if the device is tape, the data is intact, but if the device is a printer, the first character of the overflow line is not printed but is treated instead as the print control. Unless the user has planned the overflow, the first character would be random and thus the overflow print line control can be any of the posible ones. It is suggested that when the device is a printer, the option be changed to provide single spacing.

3. It is not considered an error if the END parameter is present in a READ statement. No message or traceback is printed and the error count is not altered.

4. For an input/output error, the buffer may have been partially filled or not filled at all when the error was detected. Thus,

the buffer contents could be blank when printed. When an ERR parameter is specified in a READ statement, it is honored even though the error occurrence is greater than the amount allowed.

5. The count field does not necessarily mean that up to 10 missing DD cards will be detected in a single debugging run, since a single WRITE performed in a loop could cause 10 occurrences of the message for the same missing DD card.

MESSAGE CORRECTIVE ACTION CROSS REFERENCE TABLES

Error Code	Parameters Passed To User	Standard Corrective Action	User-Supplied Corrective Action
205	A,B,D	Program termination.	See Note 1.
206	A,B,I	I=low order part of number for input too large.	User may alter I (see Note 2).
211	A,B,C	Treat format field containing C as end of FORMAT statement.	If compiled FORMAT statement, put hexadecimal equivalent of character in C. If variable format, move EBCDIC character into C. (See Note 3).
212	A,B,D	Input: Ignore remainder of I/O list. Output: Continue by starting new output record. Supply carriage control character if required by Option Table.	See Note 1.
213	A,B,D	Ignore remainder of I/O list.	See Note 1.
214	A,B,D	Input: Ignore remainder of I/O list. Ignore input/output request if for ASCII tape. Output: If unformatted write initially requested, change record format to VS. If formatted write initially requested, ignore input/output request.	If user correction is requested, the remainder of the I/O list is ignored.
215	A,B,E	Substitute zero for the invalid character.	The character placed in E will be substituted for the invalid character; input/output operations may not be performed (see Note 3).
217	A,B,D	Increment FORTRAN sequence number and read next file.	See Note 1.
218	A,B,D,F	Ignore remainder of I/O list.	See Note 1.
219-224	A,B,D	Ignore remainder of I/O list.	See Note 1.

Figure 25 (Part 1 of 2). Corrective Action after Error

Error Code	Parameters Passed To User	Standard Corrective Action	User-Supplied Corrective Action
225	A,B,E	Substitute zero for the invalid character.	The character placed in E will be substituted for the invalid character (see Note 3).
226	A,B,R	R=0 for input number too small. R = 16**63 - 1 for input number too large.	User may alter R.
227	A,B,D	Ignore remainder of I/O list.	See Note 1.
229	A,B,D	Ignore remainder of I/O list.	See Note 1.
231	A,B,D	Ignore remainder of I/O list.	See Note 1.
232	A,B,D,G	Ignore remainder of I/O list.	See Note 1.
233	A,B,D	Change record number to list maximum allowed (32000).	See Note 1.
234 236	A,B,D	Ignore remainder of I/O list.	See Note 1.
237	A,B,D,F	Ignore remainder of I/O list.	See Note 1.
238	A,B,D	Ignore remainder of I/O list.	See Note 1.
240	See Note 4	Program termination	None
286	A,B,D	Ignore request	See Note 1.
287	A,B,D	Ignore request	see Note 1.
288	A,B,D	Implied wait	See Note 1.

Figure 25 (Part 2 of 2). Corrective Action after Error

Notes to Figure 25:

Parameter	Meaning
A	Address of return code field (INTEGER*4)
B	Address of error number (INTEGER*4)
C	Address of invalid format character (LOGICAL*1)
D	Address of data set reference number (INTEGER*4)
E	Address of invalid character (LOGICAL*1)
F	Address of DECB
G	Address of record number requested (INTEGER*4)
I	Result after conversion (INTEGER*4)
R	Result after conversion (REAL*4)

If error condition 218 (input/output error detected) occurs while error messages are being written to the object error data set, the message is written to the console and the job is terminated. If no DD card has been supplied for the object error data set, error message IFY219I is written out at the console and the job is terminated.

Note 1: If the error was not caused during asynchronous input/output processing, the user exit-routine cannot perform any asynchronous I/O operation and, in addition, may not perform any REWIND, BACKSPACE, or ENDFILE operation. If the error was caused during asynchronous input/output processing, the user cannot perform any input/output operation. On return to the library, the remainder of the input/output request will be ignored.

Note 2: The user exit routine may supply an alternative answer for the setting of the result register. The routine should always set an INTEGER*4 variable and the FORTRAN library will load fullword or halfword depending on the length of the argument causing the error.

Note 3: Alternatively, the user can set the return code to 0, thus requesting a standard corrective action.

Note 4: Code 240 generates a message showing the system or program code causing program termination, the address of the STAE Control Block, and the contents of the last PSW when abnormal termination occurred. Further information appears under message code IFY240 in VS FORTRAN System Service Reference Supplement.

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options	
			Standard Corrective Action (See Notes 2 and 3)	User-supplied Corrective Action (See Note 4)
241	K=I**J	I=0, J≤0	K=0	I,J
242	Y=X**I	X=0, I≤0	If I=0, Y=1 If I<0, Y=•	X,I
243	DA=D**I	D=0, I≤0	If I=0, Y=1 If I<0, Y=•	D,I
244	XA=X**Y	X=0, Y≤0	XA=0	X,Y
245	DA=D**DB	D=0, DB≤0	DA=0	D,DB
246	CA=C**I	C=0+0i, I≤0	If I=0, C=1+0i If I<0, C=•+0i	C,I
247	CDA=CD*I	C=0+0i, I≤0	If I=0, C=1+0i If I<0, C=•+0i	CD,I
248	Q=QA**J	QA=0, J≤0	J<0, Q=• J=0, Q=1	QA,J
249	Q=QA**QB	QA=0, QB≤0	QB<0, Q=• QB=0, Q=1	QA,QB
		QA<0, QB≠.0	Q=|QA|**QB	
250	Q=QA**QB	logs(QA)*QB≥252	Q=•	QA,QB
251	Y=SQRT (X)	X<0	$Y=\|X\|^{1/2}$	X
252	Y=EXP (X)	X>174.673	Y•	X
253	Y=ALOG (X)	X=0 X<0	Y=-• Y=log|X|	X X
	Y=ALOG10 (X)	X=0 X<0	Y=-• $Y=\log_{10}\|X\|$	X X
254	Y=COS (X) Y=SIN (X)	$\|X\|\geq(2^{18})*\pi$	$Y=\sqrt{2}/2$	
255	Y=ATAN2 (X,XA)	X=0. XA=0	Y=0	X,XA
256	Y=SINH (X) Y=COSH(X)	|H|≥175.366	Y=(SIGN of X)• Y=•	X
257	Y=ARSIN (X)	|X|>1	If X>1.0,ARSIN(X)=$\frac{\pi}{2}$ If X<-1.0,ARSIN(X)=$-\frac{\pi}{2}$	
	Y=ARCOS (X)		If X>1.0,ARCOS=0 If X<-1.0,ARCOS=π	
258	Y=TAN (X) Y=COTAN (X)	$\|X\|\geq(2^{18})*\pi$	Y=1	

Figure 26 (Part 1 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options: Standard Standard Corrective Action (See Notes 2 and 3)	Options: User-supplied Corrective Action (See Note 4)
259	Y=TAN (X)	X is too close to an odd multiple of $\frac{\pi}{2}$	Y=•	X
	Y=COTAN (X)	X is too close to a multiple of π	Y=•	
260	Q=2**QA	QA>252	Q=•	QA
261	DA=DSQRT (D)	D<0	DA=$\|D\|^{1/2}$	D
262	DA=DEXP (D)	D>174.673	D=•	D
263	DA=DLOG (D)	D=0 D<0	DA=-• DA=log|X|	D
	DA=DLOG10 (D)	D=0 D<0	DA=-• DA=$\log_{10}$ |X|	D
264	DA=DSIN (D) DA=DCOS (D)	$\|D\| \geq (2^{50}) * \pi$	DA=$\sqrt{2}/2$	D
265	DA=DATAN2(D,DB)	D=0,DB=0	DA=0	D,DB
266	DA=DSINH (D) DA=DCOSH (D)	|D|≥175.366	DA=(SIGN of X)• DA=•	D
267	DA=DARSIN (D)	|D|>1	If D>1.0 DARSIN = $\frac{\pi}{2}$ If D<-1.0 DARSIN=-$\frac{\pi}{2}$	
	DA=DARCOS (D)		If D>1.0 DARCOS (D)=0 If D<-1.0 DARCOS(D)=π	
268	DA=DTAN (D) DA=DCOTAN (D)	$\|X\| \geq (2^{50}) * \pi$	DA=1	D
269	DA=DTAN (D)	D is too close to an odd multiple of $\frac{\pi}{2}$	DA=•	D
	DA=DCOTAN (D)	D is too close to a multiple of π	DA=•	D
For error 270, CQA=X_1+iX_2				
270	CQ=CQA**J	CQA=0+0i J≤0	J=0, CQ=1+0.i J<0, CQ=•+0.i	CQA,J
For errors 271 through 275, C=X_1+iX_2				
271	Z=CEXP (C)	X^1>174.673	Z=*(COS X_2 + iSIN X_2)	C
272	Z=CEXP (C)	$\|X_2\| \geq (2^{18}) * \pi$	Z=e^{X_1}+0*i	C
273	Z=CLOG (C)	C=0+0i	Z=-•+0i	C

Figure 26 (Part 2 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options: Standard Corrective Action (See Notes 2 and 3)	Options: User-Supplied Corrective Action (See Note 4)
274	Z=CSIN (C) Z=CCOS (C)	$\|X_1\| \geq (2^{18}) * \pi$	Z=0+SINH(X_2)*i Z=COSH(X_2)+0*i	C C
275	Z=SCIN (C)	$X_2 > 174.673$	$Z=\frac{\bullet}{2}$(SIN X_1+iCOS X_1)	C
	Z=CCOS (C)		$Z=\frac{\bullet}{2}$(COS X_1-iSIN X_1)	
	Z=CSIN (C)	$X_2 < -174.673$	$Z=\frac{\bullet}{2}$(SIN X_1-iCOS X_1	C
	Z=CCOS (C)		$Z=\frac{\bullet}{2}$(COS X_1+iSIN X_1)	C
For errors 276 through 280, CQ=X_1+iX_2				
276	Z=CQEXP (CQ)	$X_1 > 174.673$	Z=*(COS X_2+iSIN X_2)	CQ
277	Z=CQEXP (CQ)	$\|X_2\| > 2^{100}$	$Z=e^{X_1}$ +0*i	CQ
278	Z=CQLOG (CQ)	CQ=0+0i	Z=-•+0i	CQ
279	Z=CQSIN (CQ) Z=CQCOS (CQ)	$\|X_1\| \geq 2^{100}$	Z=0+DSINH(X^2)*i Z=DCOSH(X^2)+0*i	CQ
280	Z=CQSIN (CQ)	$X_2 > 174.673$	$Z=\frac{\bullet}{2}$(SIN X_1+iCOS X_1)	CQ
	Z=CQCOS (CQ)		$Z=\frac{\bullet}{2}$(COS X_1=iSIN X_1)	CQ
	Z=CQSIN (CQ)	$X_2 < -174.673$	$Z=\frac{\bullet}{2}$(SIN X_1-iCOS X_1)	CQ
	Z=CQCOS (CQ)		$Z=\frac{\bullet}{2}$(COS X_1=iSIN X_1)	
For errors 281 through 285, CD=X_1+iX_2				
281	Z=CDEXP (CD)	$X_1 > 174.673$	Z=*(COS X_2+iSIN X_2)	CD
282	Z=CDEXP (CD)	$\|X_2\| \geq (2^{50}) * \pi$	$Z=e^{X_1}$ +0*i	CD
283	Z=CDLOG (CD)	CD=0+0i	Z=-•+0i	CD
284	Z=CDSIN (CD)	$\|X_1\| \geq (2^{50}) * \pi$	Z= 0+SINH(X_2)*i	CD
	Z=CDCOS (CD)		Z= COSH(X_2)+0*i	
285	Z=CDSIN (CD)	$X_2 > 174.673$	$Z=\frac{\bullet}{2}$(SIN X_1+iCOS X_1)	CD
	Z=CDCOS (CD)		$Z=\frac{\bullet}{2}$(COS X_1-iSIN X_1)	CD
	Z=CDSIN (CD)	$X_2 < -174.673$	$Z=\frac{\bullet}{2}$(SIN X_1-iCOS X_1)	CD
	Z=CDCOS (CD)		$Z=\frac{\bullet}{2}$(COS X_1+iSIN X_1)	CD

Figure 26 (Part 3 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options	
			Standard Corrective Action (See Notes 2 and 3)	User-Supplied Corrective Action (See Note 4)
289	QA=QSQRT (Q)	Q<0	QA=$\|Q\|^{1/2}$	Q
290	Y=GAMMA (X)	X≤2^{-252} or X≥57.5744	Y=•	X
291	Y=ALGAMA (X)	X≤0 or X≥$4.2937*10^{73}$	Y=•	X
292	QA=QEXP (Q)	Q>174.673	QA=•	Q
293	QA=QLOG (Q)	Q=0 Q<0	QA=-• QA=log|X|	Q
	QA=QLOG10 (Q)	Q=0 Q<0	QA=• QA=$\log_{10}\|X\|$	Q Q
294	QA=QSIN (Q) QA=QCOS (Q)	|Q|≥2^{100}	QA=$\sqrt{2}/2$	Q
295	QA=QATAN2(Q,QB)	Q=0, QB=0	QA=0	Q,QB
296	QA=QSINH (Q) QA=QCOSH (Q)	|Q|≥175.366	QA=•(SIGN Q)• QA=•	Q
297	QA=QARSIN (Q)	|Q|>1	If Q>1.0 QARSIN=$\frac{\pi}{2}$	Q
			If Q<-1.0 QARSIN=-$\frac{\pi}{2}$	Q
	QA=QARCOS (Q)		If Q>1.0 QARCOS(Q)=0 If Q<-1.0 QARCOS(Q)=π	Q
298	QA=QTAN (Q) QA=QCOTAN (Q)	|Q|>2^{100}	QA=1	Q
299	QA=QTAN (Q)	Q is too close to an odd multiple of $\frac{\pi}{2}$	QA=•	Q
	QA=QCOTAN (Q)	Q is too close to a multiple of π	QA=•	Q
300	DA=DGAMMA (D)	D≤2^{-252} or D≥57.5774	DA=•	D
301	DA-DLGAMA (D)	D≤0 or D≥$4.2937*10^{73}$	DA=•	

Figure 26 (Part 4 of 4). Corrective Action after Mathematical Subroutine Error

Notes to Figure 26:

1. The variable types are as follows:

Variable	Type
I,J,K	INTEGER*4
X,XA,Y	REAL*4
D,DA,DB	REAL*8
C,CA	COMPLEX*8
Q,QA,QB	REAL*16
CQ,CQA	COMPLEX*32
X,X ,X	Complex variables to be given the length of the functioned argument when they appear.
CD,CDA	COMPLEX*16

2. The largest number that can be represented in floating point in indicated by the symbol •.

3. The value e=2.7183 (approximately).

4. The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.

Program Interrupt Messages			Options	
Error Code	Parameters Passed to User Exit (Note 1)	Reason for Interrupt (Note 2)	Standard Corrective Action	User-Supplied Corrective Action
207	D,I	Exponent overflow (Interrupt Code 12)	Result register set to the largest possible floating point number. The sign of the result register is not altered.	User may alter D. (Note 3)
208	D,I	Exponent underflow (Interrupt Code 13)	The result register is set to zero.	User may alter D. (Note 3)
209	None	Divide check, integer divide (interrupt code 9), decimal divide (Interrupt Code 11), floating point Code 11), floating point divide (interrupt code 15). See Note 4.	For floating point divide, where n/0 and n=0, result register is set to 0; where n≠0, result register set to largest possible floating point number. No standard fixup for other interrupts.	See Note 5.
210	None	Specification interrupt (interrupt Code 6) occurs for boundary misalignment. Operation exception (interrupt code 1) occurs for operation interrupt. Other interrupts occur during boundary alignment adjustment or extended precision floating point simulation. They will be shown with this error code and the PSW portion of the message will identify the interrupt.	No special corrective action other than correcting boundary misalignments.	See Note 5.

Figure 27. Corrective Action after Program Interrupt

Notes to Figure 27:

1. The variable types and meaning are as follows:

Variable	Type	Meaning
D	REAL*8	This variable contains the contents
I	INTEGER*4	The variable contains the "exponent" as an integer value for the number in D. It may be used to determine the amount of the underflow or overflow. The value in I is not the true exponent, but what was left in the exponent field of a floating point number after the interrupt.

2. Asynchronous Program interrupts are described in the appropriate principles of operation publiction, as listed in the Preface.

3. The user exit routine may supply an alternate answer for the setting of the result register. This is accomplished by

placing a value for D in the user-exit routine. Although the interrupt may be caused by a long or short floating-point operation, the user-exit routine need not be concerned with this. The user-exit routine should always set a REAL*16 variable and the FORTRAN library will load the correct length data item depending upon the floating-point operation that caused the interrupt.

4. For floating-point divide check, the contents of the result register is shown in the message.

5. The user-exit routine does not have the ability to change result registers after a fixed-point divide check. The boundary alignment adjustments are informative messages, and there is nothing to alter before execution continues.

DVCHK SUBROUTINE

The CALL DVCHK statement tests for a divide-check exception and returns a value indicating the existing condition.

Syntax

```
CALL DVCHK (i)
```

i
is an integer or real variable in the program unit.

The values of *i* returned have the following meanings:

Value	Meaning
1	The divide-check indicator is *on*.
2	The divide-check indicator is *off*.

DUMP/PDUMP SUBROUTINE

The CALL DUMP/PDUMP statement dynamically dumps a specified area of storage.

Syntax

```
CALL [DUMP|PDUMP] (a1,b1,i1,...an,bn,in)
```

a and *b*
are each a variable in the program unit. They indicate areas of storage to be dumped.

Either *a* or *b* can represent the upper or lower limits of the storage area.

i
specifies the dump format to be used.

The values that can be specified for *i* and their meanings are:

Value	Format Requested
1	Hexadecimal
2	LOGICAL*4
3	INTEGER*2
4	INTEGER*4
5	REAL*4
6	REAL*8
7	COMPLEX*8
8	COMPLEX*16
9	CHARACTER
10	REAL*16
11	COMPLEX*32

When a CALL DUMP statement is executed, the area requested is dumped onto the system output data set and execution is terminated.

When a CALL PDUMP statement is executed, the area requested is dumped onto the system output data set and execution continues.

CDUMP/PCDUMP SUBROUTINE

The CALL CDUMP/PCDUMP statement dynamically dumps a specified area of storage.

Syntax

```
CALL [CDUMP|PCDUMP] (a1,b1,...an,bn)
```

a and *b*
are each a variable in the program unit. They indicate areas of storage to be dumped.

Either *a* or *b* can represent the upper or lower limits of the storage area.

The dump is always produced in character format.

EXIT SUBROUTINE

The CALL EXIT statement terminates execution of the load module or phase and returns control to the operating system.

Syntax

```
CALL EXIT
```

CALL EXIT performs a function similar to that of the STOP statement, except that no operator message is produced.

OPSYS SUBROUTINE (DOS ONLY)

The CALL OPSYS statement loads the overlay feature, allowing the user to divide a program into a number of phases.

Syntax

```
CALL OPSYS('LOAD','phasename')
```

LOAD
is required to be entered as shown.

'*phasename*'
specifies the name of the phase to be loaded. The phase must be in the core image library.

the '*phasename*' must be specified in eight alphameric characters. If fewer than eight characters are specified, the name should be left-adjusted within the field and padded on the right with blanks. Alternatively, the name of the phase may be specified as a variable or in an array.

OVERFLW SUBROUTINE

The CALL OVERFLW statement tests for exponent overflow or underflow, and returns a value indicating the existing condition.

Syntax

```
CALL OVERFLW (i)
```

i
is an integer or real variable defined within this program unit.

The values of j returned have the following meanings:

Value	Meaning
1	Floating-point overflow occurred last.
2	No overflow or underflow condition is current.
3	Floating-point underflow occurred last.

Note: The values for 1 and 3 indicate the last one to occur; if the same statement causes an overflow followed by an underflow the value returned is 3 (underflow occurred last).

APPENDIX E. EBCDIC AND ASCII CODES

EBCDIC refers to IBM EBCDIC code point ordering for the 256 character set.

ISO 8 bit refers to ISO 2022 code point ordering for the 256 character set.

ASCII 7 bit refers to ANSI X3.4-1977 code point ordering for the 128 character set.

ASCII 6 bit refers to ANSI X3.32-1973 code point ordering for the 64 character set.

The column used for the lexical intrinsic functions is ASCII 7 bit.

The blank character to be used to extend character strings for the intrinsic functions LGE, LGT, LLE, and LLT is the ASCII blank (HEX 20).

Note 1: This position does not exist in ANSI X3.4-1977 for 7-bit code.

Note 2: This position does not exist in ANSI X3.32-1973 for 6-bit code.

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ASCII 7 bit for ICHAR	ASCII 6 bit for ICHAR
00	0	NUL	Null	0	0	Note 2
01	1	SOH	Start of heading	1	1	Note 2
02	2	STX	Start of text	2	2	Note 2
03	3	ETX	End of text	3	3	Note 2
04	4	SEL	Select	156	Note 1	Note 2
05	5	HT	Horizontal Tab	9	9	Note 2
06	6	RNL	Required new line	134	Note 1	Note 2
07	7	DEL	Delete	127	127	Note 2
08	8	GE	Graphic Escape	151	Note 1	Note 2
09	9	SPS	Superscript	141	Note 1	Note 2
0A	10	RPT	Repeat	142	Note 1	Note 2
0B	11	VT	Vertical Tab	11	11	Note 2
0C	12	FF	Form Feed	12	12	Note 2
0D	13	CR	Carriage Return	13	13	Note 2
0E	14	SO	Shift out	14	14	Note 2
0F	15	SI	Shift in	15	15	Note 2
10	16	DLE	Data link escape	16	16	Note 2
11	17	DC1	Device control 1	17	17	Note 2
12	18	DC2	Device control 2	18	18	Note 2
13	19	DC3	Device control 3	19	19	Note 2
14	20	RES ENP	Restore Enable presentation	157	Note 1	Note 2
15	21	NL	New line acknowledgement	133	Note 1	Note 2
16	22	BS	Backspace	8	8	Note 2
17	23	POC	Program-operator communication	135	Note 1	Note 2
18	24	CAN	Cancel	24	24	Note 2
19	25	EM	End of Medium	25	25	Note 2
1A	26	UBS	Unit backspace	146	Note 1	Note 2
1B	27	CU1	Customer use 1	143	Note 1	Note 2
1C	28	IFS	Interchange file separator	28	28	Note 2
1D	29	IGS	Interchange group separator	29	29	Note 2

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ASCII 7 bit for ICHAR	ASCII 6 bit for ICHAR
1E	30	IRS	Interchange record separator	30	30	Note 2
1F	31	IUS ITB	Interchange unit separator Intermediate trans. block	31	31	Note 2
20	32	DS	Digit select	128	Note 1	Note 2
21	33	SOS	Start of significance	129	Note 1	Note 2
22	34	FS	Field separator	130	Note 1	Note 2
23	35	WUS	Word underscore	131	Note 1	Note 2
24	36	BYP INP	Bypass Inhibit presentation	132	Note 1	Note 2
25	37	LF	Line feed	10	10	Note 2
26	38	ETB	End of trans. block	23	23	Note 2
27	39	ESC	Escape	27	27	Note 2
28	40		Reserved	136	Note 1	Note 2
29	41		Reserved	137	Note 1	Note 2
2A	42	SM, SW	Set mode, Switch	138	Note 1	Note 2
2B	43	FMT	Format	139	Note 1	Note 2
2C	44		Reserved	140	Note 1	Note 2
2D	45	ENQ	Enquiry	5	5	Note 2
2E	46	ACK	Acknowledge	6	6	Note 2
2F	47	BEL	Bell	7	7	Note 2
30	48		Reserved	144	Note 1	Note 2
31	49		Reserved	145	Note 1	Note 2
32	50	SYN	Synchronous	22	22	Note 2
33	51	IR	Index	147	Note 1	Note 2
34	52	PP	Presentation position	148	Note 1	Note 2
35	53	TRN	Transparent	149	Note 1	Note 2
36	54	NBS	Numeric backspace	150	Note 1	Note 2
37	55	EOT	End of transmission	4	4	Note 2
38	56	SBS	Subscript	152	Note 1	Note 2
39	57	IT	Indent	153	Note 1	Note 2
3A	58	RFF	Required	154	Note 1	Note 2
3B	59	CU3	Customer use 3	155	Note 1	Note 2
3C	60	DC4	Device code 4	20	20	Note 2
3D	61	NAK	Negative acknowledge	21	21	Note 2
3E	62		Reserved	158	Note 1	Note 2
3F	63	SUB	Substitute	26	26	Note 2
40	64	SP	Space	32	32	0
41	65	RSP	Required space	160	Note 1	Note 2
42	66			161	Note 1	Note 2
43	67			162	Note 1	Note 2
44	68			163	Note 1	Note 2
45	69			164	Note 1	Note 2
46	70			165	Note 1	Note 2
47	71			166	Note 1	Note 2
48	72			167	Note 1	Note 2
49	73			168	Note 1	Note 2
4A	74	¢	Cent sign	91	91	59
4B	75	.	Period, decimal point	46	46	14
4C	76	<	Less-than sign	60	60	28
4D	77	(	Left parenthesis	40	40	8
4E	78	+	Plus sign	43	43	11
4F	79	|	Logical OR	33	33	1

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ASCII 7 bit for ICHAR	ASCII 6 bit for ICHAR
50	80	&	Ampersand	38	38	6
51	81			169	Note 1	Note 2
52	82			170	Note 1	Note 2
53	83			171	Note 1	Note 2
54	84			172	Note 1	Note 2
55	85			173	Note 1	Note 2
56	86			174	Note 1	Note 2
57	87			175	Note 1	Note 2
58	88			176	Note 1	Note 2
59	89			177	Note 1	Note 2
5A	90	!	Exclamation point	93	93	61
5B	91	$	Currency symbol	36	36	4
5C	92	*	Asterisk	42	42	10
5D	93	)	Right parenthesis	41	41	9
5E	94	;	Semicolon	59	59	27
5F	95	¬	Logical NOT	94	94	62
60	96	-	Minus sign, Hyphen	45	45	13
61	97	/	Slash	47	47	15
62	98			178	Note 1	Note 2
63	99			179	Note 1	Note 2
64	100			180	Note 1	Note 2
65	101			181	Note 1	Note 2
66	102			182	Note 1	Note 2
67	103			183	Note 1	Note 2
68	104			184	Note 1	Note 2
69	105			185	Note 1	Note 2
6A	106	|	Vertical line	124	124	Note 2
6B	107	,	Comma	44	44	12
6C	108	%	Percent sign	37	37	5
6D	109	_	Underscore	95	95	63
6E	110	>	Greater-than sign	62	62	30
6F	111	?	Question mark	63	63	31
70	112			186	Note 1	Note 2
71	113			187	Note 1	Note 2
72	114			188	Note 1	Note 2
73	115			189	Note 1	Note 2
74	116			190	Note 1	Note 2
75	117			191	Note 1	Note 2
76	118			192	Note 1	Note 2
77	119			193	Note 1	Note 2
78	120			194	Note 1	Note 2
79	121	GRA	Grave accent	96	96	Note 2
7A	122	:	Colon	58	58	26
7B	123	#	Number sign	35	35	3
7C	124	@	At sign	64	64	32
7D	125	'	Prime, Apostrophe	39	39	7
7E	126	=	Equal sign	61	61	29
7F	127	"	Quotation marks	34	34	2
80	128			195	Note 1	Note 2
81	129	a	Lower case a	97	97	Note 2
82	130	b	Lower case b	98	98	Note 2
83	131	c	Lower case c	99	99	Note 2
84	132	d	Lower case d	100	100	Note 2
85	133	e	Lower case e	101	101	Note 2
86	134	f	Lower case f	102	102	Note 2
87	135	g	Lower case g	103	103	Note 2
88	136	h	Lower case h	104	104	Note 2
89	137	i	Lower case i	105	105	Note 2
8A	138			196	Note 1	Note 2
8B	139			197	Note 1	Note 2

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ASCII 7 bit for ICHAR	ASCII 6 bit for ICHAR
8C	140			198	Note 1	Note 2
8D	141			199	Note 1	Note 2
8E	142			200	Note 1	Note 2
8F	143			201	Note 1	Note 2
90	144			202	Note 1	Note 2
91	145	j	Lower case j	106	106	Note 2
92	146	k	Lower case k	107	107	Note 2
93	147	l	Lower case l	108	108	Note 2
94	148	m	Lower case m	109	109	Note 2
95	149	n	Lower case n	110	110	Note 2
96	150	o	Lower case o	111	111	Note 2
97	151	p	Lower case p	112	112	Note 2
98	152	q	Lower case q	113	113	Note 2
99	153	r	Lower case r	114	114	Note 2
9A	154			203	Note 1	Note 2
9B	155			204	Note 1	Note 2
9C	156			205	Note 1	Note 2
9D	157			206	Note 1	Note 2
9E	158			207	Note 1	Note 2
9F	159			208	Note 1	Note 2
A0	160			209	Note 1	Note 2
A1	161	TIL	Tilde	126	126	Note 2
A2	162	s	Lower case s	115	115	Note 2
A3	163	t	Lower case t	116	116	Note 2
A4	164	u	Lower case u	117	117	Note 2
A5	165	v	Lower case v	118	118	Note 2
A6	166	w	Lower case w	119	119	Note 2
A7	167	x	Lower case x	120	120	Note 2
A8	168	y	Lower case y	121	121	Note 2
A9	169	z	Lower case z	122	122	Note 2
AA	170			210	Note 1	Note 2
AB	171			211	Note 1	Note 2
AC	172			212	Note 1	Note 2
AD	173			213	Note 1	Note 2
AE	174			214	Note 1	Note 2
AF	175			215	Note 1	Note 2
B0	176			216	Note 1	Note 2
B1	177			217	Note 1	Note 2
B2	178			218	Note 1	Note 2
B3	179			219	Note 1	Note 2
B4	180			220	Note 1	Note 2
B5	181			221	Note 1	Note 2
B6	182			222	Note 1	Note 2
B7	183			223	Note 1	Note 2
B8	184			224	Note 1	Note 2
B9	185			225	Note 1	Note 2
BA	186			226	Note 1	Note 2
BB	187			227	Note 1	Note 2
BC	188			228	Note 1	Note 2
BD	189			229	Note 1	Note 2
BE	190			230	Note 1	Note 2
BF	191			231	Note 1	Note 2
C0	192	{	Opening brace	123	123	Note 2
C1	193	A	Upper case A	65	65	33
C2	194	B	Upper case B	66	66	34
C3	195	C	Upper case C	67	67	35
C4	196	D	Upper case D	68	68	36
C5	197	E	Upper case E	69	69	37
C6	198	F	Upper case F	70	70	38
C7	199	G	Upper case G	71	71	39

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ASCII 7 bit for ICHAR	ASCII 6 bit for ICHAR
C8	200	H	Upper case H	72	72	40
C9	201	I	Upper case I	73	73	41
CA	202			232	Note 1	Note 2
CB	203			233	Note 1	Note 2
CC	204			234	Note 1	Note 2
CD	205			235	Note 1	Note 2
CE	206			236	Note 1	Note 2
CF	207			237	Note 1	Note 2
D0	208	}	Closing brace	125	125	Note 2
D1	209	J	Upper case J	74	74	42
D2	210	K	Upper case K	75	75	43
D3	211	L	Upper case L	76	76	44
D4	212	M	Upper case M	77	77	45
D5	213	N	Upper case N	78	78	46
D6	214	O	Upper case O	79	79	47
D7	215	P	Upper case P	80	80	48
D8	216	Q	Upper case Q	81	81	49
D9	217	R	Upper case R	82	82	50
DA	218			238	Note 1	Note 2
DB	219			239	Note 1	Note 2
DC	220			240	Note 1	Note 2
DD	221			241	Note 1	Note 2
DE	222			242	Note 1	Note 2
DF	223			243	Note 1	Note 2
E0	224	\	Reverse slant	92	92	60
E1	225			159	Note 1	Note 2
E2	226	S	Upper case S	83	83	51
E3	227	T	Upper case T	84	84	52
E4	228	U	Upper case U	85	85	53
E5	229	V	Upper case V	86	86	54
E6	230	W	Upper case W	87	87	55
E7	231	X	Upper case X	88	88	56
E8	232	Y	Upper case Y	89	89	57
E9	233	Z	Upper case Z	90	90	58
EA	234			244	Note 1	Note 2
EB	235			245	Note 1	Note 2
EC	236			246	Note 1	Note 2
ED	237			247	Note 1	Note 2
EE	238			248	Note 1	Note 2
EF	239			249	Note 1	Note 2
F0	240	0	Zero	48	48	16
F1	241	1	One	49	49	17
F2	242	2	Two	50	50	18
F3	243	3	Three	51	51	19
F4	244	4	Four	52	52	20
F5	245	5	Five	53	53	21
F6	246	6	Six	54	54	22
F7	247	7	Seven	55	55	23
F8	248	8	Eight	56	56	24
F9	249	9	Nine	57	57	25
FA	250	|	Long vertical mark	250	Note 1	Note 2
FB	251			251	Note 1	Note 2
FC	252			252	Note 1	Note 2
FD	253			253	Note 1	Note 2
FE	254			254	Note 1	Note 2
FF	255	EO	Eight ones	255	Note 1	Note 2

GLOSSARY

This glossary includes definitions developed by the American National Standards Institute (ANSI) and the International Organization for Standarization (ISO).

An asterisk (*) to the left of a term indicates that the entire entry is reproduced from the American National Dictionary for Information Processing, copyright 1977 by the Computer and Business Equipment Manufacturers Association, copies of which may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

An asterisk (*) to the right of an item number indicates an ANSI definition in an entry that also includes other definitions.

The symbol "(ISO)" at the beginning of a definition indicates that it has been discussed and agreed upon at meetings of the International Organization for Standardization Technical Committee 97/Subcommittee 1 (Data Processing Vocabulary), and has also been approved by ANSI and included in the American National Dictionary for Information Processing.

alphabetic character. A character of the set A, B, C,...,Z. See also "letter."

IBM EXTENSION

In VS FORTRAN, the currency symbol ($) is considered an alphabetic character.

END OF IBM EXTENSION

alphameric. Pertaining to a character set that contains letters, digits, and other characters, such as punctuation marks.

alphameric character set. A character set that contains both letters and digits and also contains control characters, special characters, and the space character.

argument. A parameter passed between a calling program and a SUBROUTINE subprogram, a FUNCTION subprogram, or a statement function.

arithmetic constant. A constant of type integer, real, double precision, or complex.

arithmetic expression. One or more arithmetic operators and/or arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, array element, or function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator. A symbol that directs VS FORTRAN to perform an arithmetic operation. The arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation.

array. An ordered set of data items identified by a single name.

array declarator. The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension. An array declarator may appear in a DIMENSION, COMMON, or explicit type statement.

array element. A data item in an array, identified by the array name followed by a subscript indicating its position in the array.

array name. The name of an ordered set of data items that make up an array.

assignment statement. A statement that assigns a value to a variable or array element. It is made up of a variable or array element, followed by an equal sign (=), followed by an expression. The variable, array element, or expression can be character, logical, or arithmetic. When the assignment statement is executed, the expression to the right of the equal sign replaces the value of the variable or array element to the left.

basic real constant. A string of decimal digits containing a decimal point, and expressing a real value.

blank common. An unnamed common block.

character constant. A string of one or more alphameric characters enclosed in apostrophes. The delimiting apostrophes are not part of the constant.

character expression. An expression in the form of a single character constant, variable, array element, substring, function reference, or another expression enclosed in parentheses. A character expression is always of type character.

character type. A data type that can consist of any alphameric characters; in storage, one byte is used for each character.

common block. A storage area that may be referred to by a calling program and one or more subprograms.

complex constant. An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first real constant of the pair is the real part of the complex number; the second is the imaginary part.

complex type. An approximation of the value of a complex number, consisting of an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

connected file. A file that has been connected to a unit and defined by a FILEDEF command or by job control statements.

constant. An unvarying quantity. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and hexadecimal data.

control statement. Any of the statements used to alter the normal sequential execution of FORTRAN statements, or to terminate the execution of a FORTRAN program. FORTRAN control statements are any of the forms of the GO TO, IF, and DO statements, or the PAUSE, CONTINUE, and STOP statements.

data. (1)* (ISO) A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. (2) In FORTRAN, data includes constants, variables, arrays, and character substrings.

data item. A constant, variable, array element, or character substring.

data set. The major unit of data storage and retrieval consisting of data collected in one of several prescribed arrangements and described by control information to which the system has access.

data set reference number. A constant or variable in an input or output statement that identifies a data set to be processed.

data type. The properties and internal representation that characterize data and functions. The basic types are integer, real, complex, logical, double precision, and character.

*** digit.** (ISO) A graphic character that represents an integer. For example, one of the characters 0 to 9.

DO loop. A range of statements executed repetitively by a DO statement. See also "range of a DO."

double precision. The standard name for real data of storage length 8.

DO variable. A variable, specified in a DO statement, that is initialized or incremented prior to each execution of the statement or statements within a DO range. It is used to control the number of times the statements within the range are executed. See also "range of a DO."

dummy argument. A variable within a subprogram or statement function definition with which actual arguments from the calling program or function reference are positionally associated. Dummy arguments are defined in a SUBROUTINE or FUNCTION statement, or in a statement function definition.

executable program. A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures, or both.

executable statement. A statement that causes an action to be taken by the program; for example, to calculate, to test conditions, or to alter the flow of control.

existing file. A file that has been defined by a FILEDEF command or by job control statements.

expression. A notation that represents a value: a constant or a reference appearing alone, or combinations of constants and/or references with operators. An expression can be arithmetic, character, logical, or relational.

external file. A set of related external records treated as a unit; for example, in stock control, an external file would consist of a set of invoices.

external function. A function defined outside the program unit that refers to it.

external procedure. A SUBROUTINE or FUNCTION subprogram written in FORTRAN.

file. A set of records. If the file is located in internal storage, it is an internal file; if it is on an input/output device, it is an external file.

file definition statement. A statement that describes the characteristics of a file to a user program. For example, the OS/VS DD statement or DOS/VSE ASSGN

statement for batch processing, or the FILEDEF command for CMS processing.

file reference. A reference within a program to a file. It is specified by a unit identifier.

formatted record. (1) A record, described in a FORMAT statement, that is transmitted, when necessary with data conversion, between internal storage and internal storage or to an external record. (2) A record transmitted with list-directed READ or WRITE statements and an EXTERNAL statement.

FORTRAN-supplied procedure. See "intrinsic function."

function reference. A source program reference to an intrinsic function, to an external function, or to a statement function.

function subprogram. A subprogram invoked through a function reference, and headed by a FUNCTION statement. It returns a value to the calling program unit at the point of reference.

IBM EXTENSION

hexadecimal constant. A constant that is made up of the character Z followed by two or more hexadecimal digits.

END OF IBM EXTENSION

hierarchy of operations. The relative order used to evaluate expressions containing arithmetic, logical, or character operations.

implied DO. An indexing specification (similar to a DO statement, but without specifying the word DO) with a list of data elements, rather than a set of statements, as its range. In a DATA statement the list can contain integer constants or expressions containing integer constants. In input/output statements the list can contain integer, real, or double precision arithmetic expressions.

integer constant. A string of decimal digits containing no decimal point and expressing a whole number.

integer expression. An arithmetic expression whose values are of integer type.

integer type. An arithmetic data type capable of expressing the value of an integer. It can have a positive, negative, or zero value. It must not include a decimal point.

internal file. A set of related internal records treated as a unit.

intrinsic function. A function, supplied by VS FORTRAN, that performs mathematical or character operations.

*** I/O.** Pertaining to either input or output, or both.

I/O list. A list of variables in an input or output statement specifying which data is to be read or which data is to be written. An output list may also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

labeled common. See "named common."

length specification. A source language specification of the number of bytes to be occupied by a variable or an array element.

letter. A symbol representing a unit of the alphabet.

list-directed. An input/output specification that uses a data list instead of a FORMAT specification.

logical constant. A constant that can have one of two values: true or false.

logical expression. A combination of logical primaries and logical operators. A logical expression can have one of two values: true or false.

logical operator. Any of the set of operators .NOT. (negation), .AND. (connection: both), or .OR. (inclusion: either or both), .EQV. (equal), .NEQV. (not equal).

logical primary. A primary that can have the value true or false. See also "primary."

logical type. A data type that can have the value true or false for VS FORTRAN. See also "data type."

looping. Repetitive execution of the same statement or statements. Usually controlled by a DO statement.

main program. A program unit, required for execution, that can call other program units but cannot be called by them.

name. A string of from one through six alphameric characters, the first of which must be alphabetic. Used to identify a constant, a variable, an array, a function, a subroutine, or a common block.

named common. A separate common block consisting of variables, arrays, and array declarators, and given a name.

nested DO. A DO statement whose range is entirely contained within the range of another DO statement.

nonexecutable statement. A statement that describes the characteristics of the program unit, of data, of editing

information, or of statement functions, but does not cause an action to be taken by the program.

nonexisting file. A file that has not been defined by a FILEDEF command or by job control statements.

*** numeric character.** (ISO) Synonym for digit.

numeric constant. A constant that expresses an integer, real, or complex number.

preconnected file. A unit or file that was defined at installation time. However, a preconnected file does not exist for a program if the file is not defined by a FILEDEF command or by job control statements.

predefined specification. The implied type and length specification of a data item, based on the initial character of its name in the absence of any specification to the contrary. The initial characters I-N type data items as integer; the initial characters A-H, O-Z, and $ type data items as real. No other data types are predefined. For VS FORTRAN, the length for both types is 4 bytes.

primary. An irreducible unit of data; a single constant, variable, array element, function reference, or expression enclosed in parentheses.

procedure. A sequenced set of statements that may be used at one or more points in one or more computer programs, and that usually is given one or more input parameters and returns one or more output parameters. A procedure consists of subroutines, function subprograms, and intrinsic functions.

procedure subprogram. A function or subroutine subprogram.

program unit. A sequence of statements constituting a main program or subprogram.

range of a DO. Those statements that physically follow a DO statement, up to and including the statement specified by the DO statement as being the last to be executed; also called a "DO loop."

real constant. A string of decimal digits that expresses a real number. A real constant must contain either a decimal point or a decimal exponent and may contain both.

real type. An arithmetic data type, capable of approximating the value of a real number. It can have a positive, negative, or zero value.

record. A collection of related items of data treated as a unit.

relational expression. An expression that consists of an arithmetic expression, followed by a relational operator, followed by another arithmetic expression or a character expression, followed by a relational operator, followed by another character expression. The result is a value that is true or false.

relational operator. Any of the set of operators:

.GT.	greater than
.GE.	greater than or equal to
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to

scale factor. A specification in a FORMAT statement that changes the location of the decimal point in a real number (and, on input, if there is no exponent, the magnitude of the number).

specification statement. One of the set of statements that provides the compiler with information about the data used in the source program. In addition, the statement supplies the information required to allocate data storage.

specification subprogram. A subprogram headed by a BLOCK DATA statement and used to initialize variables in named common blocks.

statement. The basic unit of a FORTRAN program, that specifies an action to be performed, or the nature and characteristics of the data to be processed, or information about the program itself. Statements fall into two broad classes: executable and nonexecutable.

statement function. A name, followed by a list of dummy arguments, that is equated to an arithmetic, logical, or character expression. In the remainder of the program the name can be used as a substitute for the expression.

statement function definition. A statement that defines a statement function. Its form is a name, followed by a list of dummy arguments, followed by an equal sign (=), followed by an arithmetic, logical, or character expression.

statement function reference. A reference in an arithmetic, logical, or character expression to the name of a previously defined statement function.

statement label. See "statement number."

statement number. A number of from one through five decimal digits that is used to identify a statement. Statement numbers can be used to transfer control,

to define the range of a DO, or to refer to a FORMAT statement.

subprogram. A program unit that is invoked by another program unit in the same program. In FORTRAN, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

subroutine subprogram. A subprogram whose first statement is a SUBROUTINE statement. It optionally returns one or more parameters to the calling program unit.

*** subscript.** (1) (ISO) A symbol that is associated with the name of a set to identify a particular subset or element.

(2) A subscript quantity or set of subscript quantities, enclosed in parentheses and used with an array name to identify a particular array element.

subscript quantity. A component of a subscript: an integer constant, an integer variable, or an expression evaluated as an integer constant.

IBM EXTENSION

In VS FORTRAN, a subscript quantity may also be a real constant, variable, or expression.

END OF IBM EXTENSION

type declaration. The explicit specification of the type of a constant, variable, array, or function by use of an explicit type specification statement.

unformatted record. A record that is transmitted unchanged between internal storage and an external record.

unit. A means of referring to a file in order to use input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

unit identifier. The number that specifies an external unit.

1. An integer expression whose value must be zero or positive. For VS FORTRAN, this integer value of length 4 must correspond to a DD name, a FILEDEF name, or an ASSGN name.
2. An asterisk (*) that corresponds on input to FT05F001 and on output to FT06F001.
3. The name of a character array, character array element, or character substring for an internal file.

variable. (1) * A quantity that can assume any of a given set of values.

(2) A data item, identified by a name, that is not a named constant, array, or array element, and that can assume different values at different times during program execution.

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X format code 103

Z format code 99
zero 127, 129

VS FORTRAN Application Programming: Language Reference (File No. S370-25) Printed in U.S.A. GC26-3986-1

GC26-3986-1

Reader's Comment Form

VS FORTRAN Application Programming:
Language Reference
GC26-3986-1

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL _____

Previous TNL _____

Previous TNL _____

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

GC26-3986-1

Reader's Comment Form

Id 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

Id and tape **Please do not staple** Fold and tape

IBM®

VS FORTRAN Application Programming: Language Reference (File No. S370-25) Printed in U.S.A. GC26-3986-1