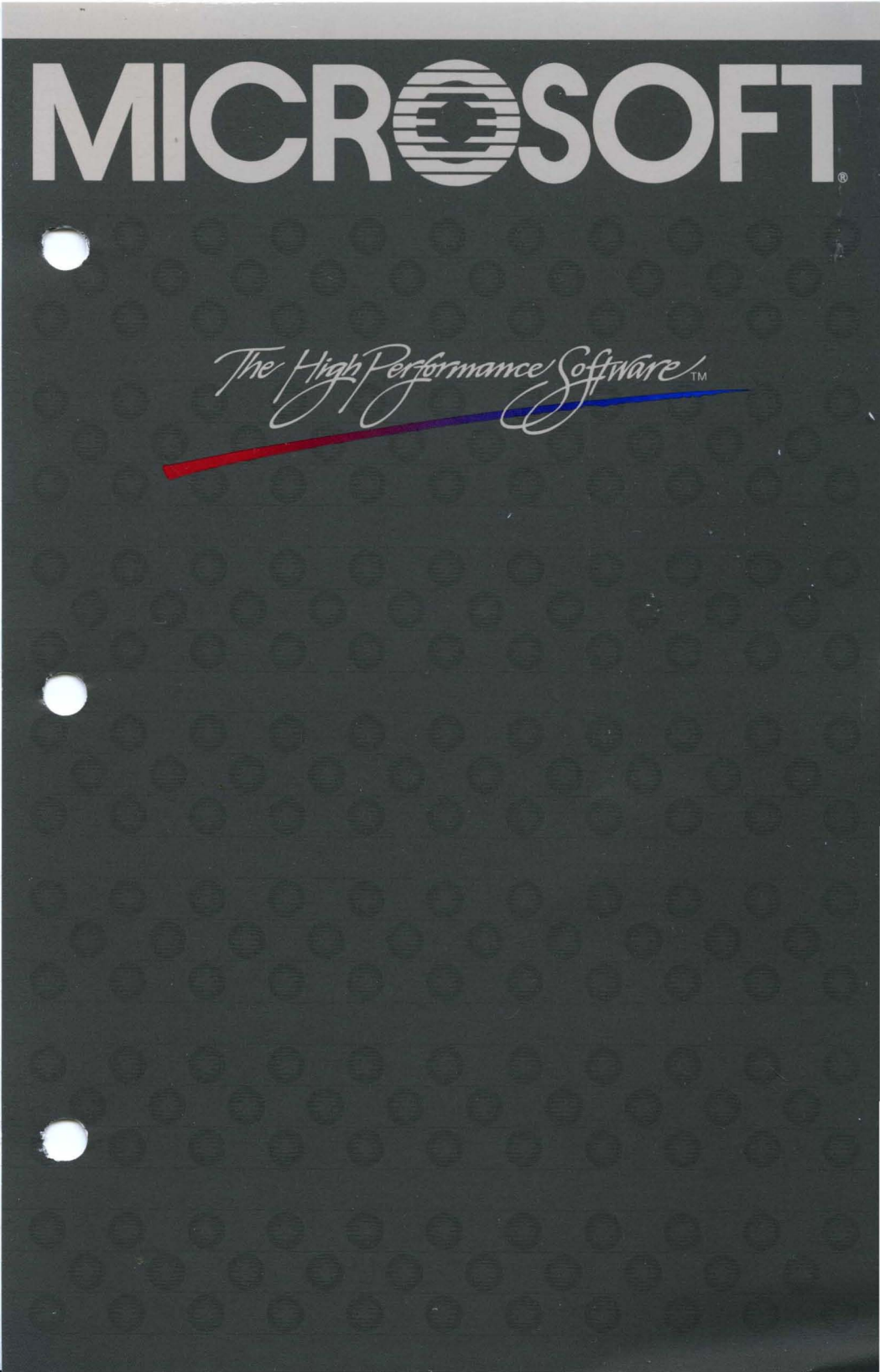

MICROSOFT®
The High Performance Software™

Microsoft® FORTRAN

Reference Manual

Microsoft Corporation

If you have comments about the software or these manuals, please complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Document Number 8205-330-09
Part Number 005-014-029

Contents

Tables

Introduction

This is a language reference manual for the Microsoft® FORTRAN language system. MS™-FORTRAN conforms to Subset FORTRAN, as described in ANSI X3.9-1978. MS-FORTRAN includes extensions to the subset language and some of the features of the full ANSI standard, commonly known as FORTRAN 77. See Appendix A, "Microsoft FORTRAN and ANSI Subset FORTRAN," for details.

The syntactical rules for using FORTRAN are rigorous and require the programmer to fully define the characteristics of the solution to a problem in a series of precise statements. Therefore, we recommend that you have a general understanding of some dialect of FORTRAN before using this product. This manual is not a tutorial; for a list of suggested FORTRAN texts, see "Learning More About FORTRAN" in this introduction.

Reference Manual Organization

Chapter 1, "Language Overview," is general in scope, providing a broad picture of the MS-FORTRAN language. Later chapters discuss the elements of the language in more detail.

Chapter 2, "Terms and Concepts," describes the smaller elements of the language, from notation to data types to expressions, and explains program structure.

Chapter 3, "Statements," defines MS-FORTRAN statements, both executable and nonexecutable.

Chapter 4, "The I/O System," provides additional information about input and output and the MS-FORTRAN file system.

Chapter 5, "Programs, Subroutines, and Functions," describes the subroutine structure, including argument passing and intrinsic (system-provided) functions.

Chapter 6, "The Microsoft FORTRAN Metacommands," describes the syntax and use of the metacommands.

Appendix A, "Microsoft FORTRAN and ANSI Subset FORTRAN," describes the differences between MS-FORTRAN and ANSI Subset FORTRAN.

Appendix B, "ASCII Character Codes," is a table of the entire ASCII character set.

Appendix C describes the structure of external Microsoft FORTRAN files.

For information on how to use the MS-FORTRAN Compiler and the details of your specific version of MS-FORTRAN, see the *Microsoft FORTRAN Compiler User's Guide.*

Descriptive Devices

The following descriptive devices are used throughout this manual to emphasize elements of the text. Descriptions of Microsoft syntax requirements for statements can be found in Chapter 3 of the *Microsoft FORTRAN Reference Manual.*

CAPS	Capitalized text indicates statements, files, or commands. The text is capitalized only to emphasize procedures, files, compilands, or objects that the user may encounter. Microsoft FORTRAN is not case sensitive. Small capitals indicate that you must press a key named by the text.
Italics	Italics indicate user-supplied data; for example, filenames, variable names, and array names.
[ ]	Square brackets indicate that the enclosed entry is optional (e.g., A[*w*] indicates that either A or A12 is valid).
...	Ellipses indicate that an entry may be repeated as many times as needed or desired. For example, the EXTERNAL statement is described as follows: EXTERNAL *name* [, *name*]... The syntactic items denoted by *name* may be repeated any number of times, separated by commas.

All other punctuation, such as commas, colons, slash marks, parentheses, and equal signs, must be entered exactly as shown.

Blanks normally have no significance in the description of MS-FORTRAN statements. The general rules for blanks, covered in Section 2.1.2, "Blanks," govern the interpretation of blanks in all contexts.

Pressing the RETURN (or ENTER) key is assumed at the end of every line you enter in response to a prompt. Only if this is the only response required is RETURN shown.

Learning More About FORTRAN

The manuals in this package provide complete reference information for your implementation of the MS-FORTRAN Compiler. They do not, however, teach you how to write programs in FORTRAN. If you are new to FORTRAN or need help in learning to program, we suggest you read any of the following books:

Agelhoff, R., and Richard Mojena. *Applied FORTRAN 77, Featuring Structured Programming.* Wadsworth, 1981.

Ashcroft, J., R.H. Eldridge, R.W. Paulson, and G. A. Wilson. *Programming With FORTRAN 77.* Granada, 1981.

Friedman, F. and E. Koffman. *Problem Solving and Structured Programming in FORTRAN.* Addison-Wesley, 2nd edition, 1981.

Wagener, J. L. *FORTRAN 77: Principles of Programming.* Wiley, 1980.

Chapter 1
Language Overview

This chapter provides a summary description of the elements of the Microsoft FORTRAN (MS-FORTRAN) language. The remaining chapters of the manual provide detailed information on these elements, from the character set to the metacommands.

1.1 Microsoft FORTRAN Metacommands

The metalanguage is the control language for the MS-FORTRAN Compiler. Metacommands let you specify options that affect the overall operation of a compilation. For example, with metacommands you can enable or disable generation of a listing file, runtime error checking code, or use of MS-FORTRAN features that are not a part of the subset or full language standard. The metalanguage consists of commands that appear in your source code, each on a line of its own and each with a dollar sign ($) in column one. The metalanguage is a level of language designed to enhance your use of the MS-FORTRAN Compiler. Although most implementations of FORTRAN have some type of compiler control, the Microsoft FORTRAN metacommands are not part of standard FORTRAN (and hence, not portable). These are the metacommands currently available:

$DEBUG	$LIST
$DECMATH	$NOLIST
$NODEBUG	$MESSAGE
$DO66	$PAGE
$FLOATCALLS	$PAGESIZE
$NOFLOATCALLS	$STORAGE
$INCLUDE	$STRICT
$LARGE	$NOTSTRICT
$NOTLARGE	$SUBTITLE
$LINESIZE	$TITLE

See Chapter 6, "The Microsoft FORTRAN Metacommands," for a complete discussion of metacommands.

1.2 Programs and Compilable Parts of Programs

The MS-FORTRAN Compiler processes program units. A program unit may be a main program, a subroutine, a function or a block data subprogram. You can compile any of these units separately and later link them together without having to recompile them as a whole.

1. Program

   Any program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement. The first statement may be a PROGRAM statement, but such a statement is not required. The execution of a program always begins with the first executable statement in the main program. Consequently, there must be one and only one main program in every executable program.

2. Subroutine

   A program unit that can be called from other program units by a CALL statement. When called, a subroutine performs the set of actions defined by its executable statements and then returns control to the statement immediately following the statement that called it. A subroutine does not directly return a value, although values can be passed back to the calling program unit via arguments or common variables.

3. Function

   A program unit referred to in an expression. A function directly returns a value that is used in the computation of that expression and in addition may pass back values via arguments. There are three kinds of functions: external, intrinsic, and statement. (Statement functions cannot be compiled separately.)

4. Block Data Subprogram

   A program unit that provides initial values for variables in COMMON blocks. Variables are normally initialized with DATA statements. Variables in COMMON may not be initialized anywhere other than in block data subprograms.

Subroutines and functions let you develop large structured programs that can be broken into parts. This is advantageous in the following situations:

1. If a program is large, breaking it into parts makes it easier to develop, test, and maintain.
2. If a program is large and recompiling the entire source file is time consuming, breaking the program into parts saves compilation time.
3. If you intend to include certain routines in a number of different programs, you can create a single object file that contains these routines and then link it to each of the programs in which the routines are used.
4. If a routine could be implemented in any of several ways, you might place it in a file and compile it separately. Then, to improve performance, you can alter the implementation, or even rewrite the routine in assembly language or in MS-Pascal, and the rest of your program will not need to change.

See Chapter 5, "Programs, Subroutines, and Functions," for a complete discussion of compilable program units.

1.3 Input/Output

Input is the transfer of data from an external medium or an internal file to internal storage. The transfer process is called reading. Output is the transfer of data from internal storage to an external medium or to an internal file. This process is called writing.

A number of statements in FORTRAN are provided specifically for the purpose of such data transfer; some I/O statements also specify that some editing of the data be performed.

In addition to the statements that transfer data, there are several auxiliary I/O statements to manipulate the external medium or to determine or describe the properties of the connection to the external medium.

The following concepts are also important for understanding the I/O system:

1. Records

   The building blocks of the FORTRAN file system. A record is a sequence of characters or values. There are three kinds of records: formatted, unformatted, and endfile.

2. Files

   Sequences of records. Files are either external or internal.

   An external file is a file on a device or a device itself. An internal file is a character variable that serves as the source or destination of some formatted I/O action.

   Files have the following properties:

   a. a filename (optional)

   b. a file position

   c. structure (formatted, unformatted, or binary)

   d. access method (sequential or direct)

   Although a wide variety of file types are possible, most applications will need just two: implicitly opened and explicitly opened external, sequential, formatted files.

See Section 3.2, "Statement Directory," for descriptions of individual I/O statements. See Chapter 4, "The I/O System," for a complete discussion of records, files, and formatted data editing.

1.4 Statements

Statements perform a number of functions, such as computing, storing the results of computations, altering the flow of control, reading and writing files, and providing information for the compiler. Statements in FORTRAN fall into two broad classes: executable and nonexecutable.

An executable statement causes an action to be performed. Nonexecutable statements do not in themselves cause operations to be performed. Instead, they specify, describe, or classify elements of the program, such as entry points, data, or program units. Table 1.2 describes the functional categories of statements.

Table 1.1

Categories of Statements in FORTRAN

Category	Description
Assignment	Executable. Assigns a value to a variable or an array element.
Comment	Nonexecutable. Allows comments within program code.
Control	Executable. Controls the order of execution of statements.
DATA	Nonexecutable. Assigns initial values to variables.
FORMAT	Nonexecutable. Provides data editing information.
I/O	Executable. Specifies sources and destinations of data transfer, and other facets of I/O operation.
Specification	Nonexecutable. Defines the attributes of variables, arrays, and programmer function names.
Statement Function	Nonexecutable. Defines a simple, locally used function.
Program Unit Heading	Nonexecutable. Defines the start of a program unit and specifies its formal arguments.

See Chapter 3, "Statements," for a complete discussion and a directory of MS-FORTRAN statements.

1.5 Expressions

An expression is a formula for computing a value. It consists of a sequence of operands and operators. The operands may contain function invocations, variables, constants, or even other expressions. The operators specify the actions to be performed on the operands.

In the following expression, plus (+) is an operator and A and B are operands:

```
A + B
```

There are four basic kinds of expressions in FORTRAN:

1. arithmetic expressions
2. character expressions
3. relational expressions
4. logical expressions

Each type of expression takes certain types of operands and uses a specific set of operators. Evaluation of every expression produces a value of a specific type.

Expressions are not statements, but may be components of statements. In the following example, the entire line is a statement; only the portion after the equal sign is an expression:

```
X = 2.0 / 3.0 + A * B
```

See Section 2.5, "Expressions," for a discussion of expressions in MS-FORTRAN.

1.6 Names

Names denote the variables, arrays, functions, or subroutines in your program, whether defined by you or by the MS-FORTRAN system. A FORTRAN name consists of a sequence of alphanumeric characters. The following restrictions apply:

1. The maximum number of characters in a name is 1320 characters (66 characters per line multiplied by twenty lines).
2. The initial character must be alphabetic; subsequent characters must be alphanumeric.
3. Blanks are skipped.
4. Only the first six alphanumeric characters are significant; the rest are ignored.

With these restrictions regarding the make-up of the name, any valid sequence of characters can be used for any FORTRAN name. There are no reserved names as in other languages.

Sequences of alphabetic characters used as keywords by the MS-FORTRAN Compiler are not confused with user-defined names. The compiler recognizes keywords by their context and in no way restricts the use of user-defined names. Thus, for example, a program can have an array named IF, READ, or GOTO, with no error (as long as it otherwise conforms to the rules that all arrays must obey). However, use of keywords for user-defined names often interferes with the readability of a program, so the practice should be avoided.

See Section 2.4, "Names," for more information on the scope and use of names in MS-FORTRAN.

1.7 Types

Data in MS-FORTRAN belongs to one of five basic types:

1. integer (INTEGER*2 and INTEGER*4)
2. single precision real (REAL*4 or REAL)
3. double precision real (REAL*8 or DOUBLE PRECISION)
4. logical (LOGICAL*2 and LOGICAL*4)
5. complex (COMPLEX*8 and COMPLEX*16)
6. character (CHARACTER)

Data types can be declared. If not declared, the type of a name is determined by its first letter (either by default or by an IMPLICIT statement). A type statement can also include dimension information.

See Section 2.3, "Data Types," for a more complete discussion of MS-FORTRAN data types. See Section 3.2, "Statement Directory," for a detailed description of the type statement.

1.8 Lines

Lines are composed of a sequence of characters. Characters beyond the 72nd on a line are ignored; lines shorter than 72 characters are assumed to be padded with blanks.

The position of characters within a line in FORTRAN is significant. Characters in columns 1 through 5 are recognized as a statement label, a character in column 6 as a continuation indicator, and characters in columns 7 through 72 as the FORTRAN statements themselves. Comments are recognized by either the letter "C" or an asterisk (*) in column 1, metacommands by a dollar sign ($) in column 1.

With some exceptions, blanks are not significant in FORTRAN. Tab characters have significance in a few circumstances, described in Section 2.1, "Notation."

Lines in MS-FORTRAN may serve as any of the following:

1. a metacommand line
2. a comment line
3. an initial line (of a statement)
4. a continuation line (of a statement)

A metacommand line has a dollar sign in column 1 and controls the operation of the MS-FORTRAN Compiler.

A comment line has either a "C", a "c", or an asterisk in column 1, or the line is entirely blank and is ignored during processing.

An initial line of a statement has either a blank or a zero in column 6 and has either all blanks or a statement label in columns 1 through 5.

A continuation line is any line that is not a metacommand line, a comment line, or an initial line, and which has blanks in columns 1 through 5, and in column 6 has a character that is not a blank or zero.

See Section 2.2, "Lines and Statements," for details on the specific uses and limitations on the several kinds of lines in MS-FORTRAN and how statements are combined to form programs and compilable parts of programs.

1.9 Characters

In the most basic sense, a FORTRAN program is a sequence of characters. When these characters are submitted to the compiler, they are interpreted in various contexts as characters, names, labels, constants, lines, and statements.

The characters used in an MS-FORTRAN source program belong to the ASCII character set, a complete listing of which is given in Appendix B, "ASCII Character Codes." Briefly, however, the character set may be divided into three groups:

1. the 52 uppercase and lowercase alphabetic characters (A through Z and a through z)
2. the 10 digits (0 through 9)
3. special characters (all other printable characters in the ASCII character set)

See Section 2.1, "Notation," for more information about the use of characters in MS-FORTRAN.

Chapter 2
Terms and Concepts

2.1 Notation

A FORTRAN source program is a sequence of ASCII characters. The ASCII character codes include:

1. 52 uppercase and lowercase letters (A through Z and a through z)
2. 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9)
3. special characters (the remaining printable characters of the ASCII character code)

2.1.1 Alphanumeric Characters

The letters and digits, treated as a single group, are called the alphanumeric characters. MS-FORTRAN interprets lowercase letters as uppercase letters in all contexts but character constants and Hollerith fields. Thus, the following user-defined names are all equivalent in MS-FORTRAN:

```
ABCDE    abcde    AbCdE    aBcDe
```

The collating sequence for the MS-FORTRAN character set is the ASCII sequence (see Appendix B, "ASCII Character Codes," for a complete table of the ASCII characters).

2.1.2 Blanks

With the exceptions noted in the following list, the blank character has no significance in an MS-FORTRAN source program and may therefore be used for improving readability. The exceptions are the following:

1. Blanks within string constants are significant.
2. Blanks within Hollerith fields are significant.
3. A blank or zero in column 6 distinguishes initial lines from continuation lines.

2.1.3 Tabs

The TAB character has the following significance in an MS-FORTRAN source program:

1. If the TAB appears in columns 1 through 5, the next character on the source line is considered to be in column 7.
2. A TAB appearing in columns 6 through 72 is considered to be a blank, even if it appears in a string or Hollerith literal.

2.1.4 Columns

The characters in a given line are positioned by columns, with the first character in column 1, the second in column 2, and so forth.

The column in which a character resides is significant in FORTRAN. Column 1 is used for comment indicators and metacommand indicators. Columns 1 through 5 are reserved for statement labels and column 6 for continuation indicators.

2.2 Lines and Statements

You can also think of a FORTRAN source program as a sequence of lines. Only the first 72 characters in a line are treated as significant by the compiler, with any trailing characters in a line ignored. Lines with fewer than 72 characters are assumed to be padded with blanks to 72 characters.

Note

This is not usually important unless the blanks are part of a Hollerith or literal string. For an illustration, see Section 2.3.6, "The Character Data Type," which describes character constants.

2.2.1 Initial Lines

An initial line is any line that is not a comment line or a metacommand line and that contains a blank or a zero character in column 6. The first five columns of the line must either be all blank or contain a label. With the exception of the statement following a logical IF, FORTRAN statements begin with an initial line.

A statement label is a sequence of one to five digits, at least one of which must be nonzero. A label may be placed anywhere in columns 1 through 5 of an initial line. Blanks and leading zeros are not significant.

2.2.2 Continuation Lines

A continuation line is any line that is not a comment line or a metacommand line and that contains any character in column 6 other than a blank or a zero. The first five columns of a continuation line must be blanks. A continuation line increases the amount of room in which to write a statement. If it will not fit on a single initial line, it may be extended to include up to 19 continuation lines.

2.2.3 Comment Lines

A line is treated as a comment line if any one of the following conditions is met:

1. a "C" (or "c") appears in column 1
2. an asterisk (*) appears in column 1
3. the line is empty or contains all blanks

Comment lines do not affect the execution of the FORTRAN program in any way. Comment lines must be followed immediately by an initial line or another comment line. They must not be followed by a continuation line.

2.2.4 Statement Definition and Order

A FORTRAN statement consists of an initial line, followed by zero to nineteen continuation lines. A statement may contain as many as 1320 characters in columns 7 through 72 of the initial line and columns 7 through 72 of the continuation lines. The END statement must be written within columns 7 through 72 of an initial line, and no other statement may have an initial line that appears to be an END statement. (In the following discussion, individual statements are simply referred to by name; see Chapter 3, "Statements," for definitions of specific statements and their properties.)

The FORTRAN language enforces a certain ordering of the statements and lines that make up a FORTRAN program unit. In addition, MS-FORTRAN enforces additional requirements in the ordering of lines and statements in an MS-FORTRAN compiland.

In general, a compiland consists of one or more program units (see Chapter 5, "Programs, Subroutines, and Functions," for more information on compilation units and subroutines). The various rules for ordering statements are illustrated in Figure 2.1 and described in the paragraphs following.

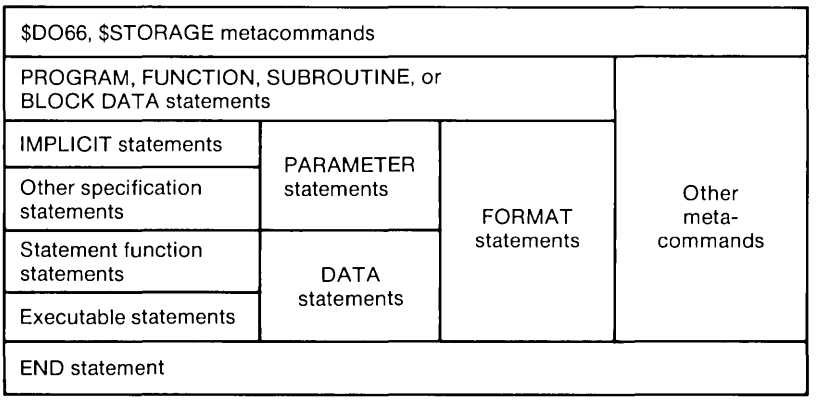


Figure 2.1. Order of Statements Within Program Units and Compilands

Within Figure 2.1, the following conventions apply:

1. Classes of lines or statements above or below other classes must appear in the designated order.
2. Classes of lines or statements may be interspersed with other classes that appear across from one another.
3. Comment lines may appear throughout the source code except before a continuation line.
4. The $LARGE and $NOTLARGE metacommands may not appear within the executable statement section.
5. BLOCK DATA subprograms may not contain statement function statements, FORMAT statements, or executable statements.

A subprogram begins with either a SUBROUTINE, FUNCTION or BLOCK DATA statement and ends with an END statement. A main program begins with a PROGRAM statement or any statement other than a SUBROUTINE, FUNCTION, or BLOCK DATA statement and ends with an END statement. A subprogram or the main program is referred to as a program unit.

Within a program unit, statements must appear in an order consistent with the following rules:

1. A PROGRAM statement, if present, or a SUBROUTINE, FUNCTION, or BLOCK DATA statement must be the first statement of the program unit.
2. FORMAT statements may appear anywhere after the SUBROUTINE or FUNCTION statement, or PROGRAM statement, if present.
3. Specification statements that define the type of a symbolic constant must appear before the PARAMETER statement that defines the value of that symbolic constant. The PARAMETER statement that defines a symbolic constant must appear before any other statement that uses that symbolic constant. Otherwise, a PARAMETER statement may appear anywhere in the specification statements.
4. The IMPLICIT statement must precede all specification statements except the PARAMETER statement. All specification statements must precede all DATA statements, statement function statements, and executable statements.

5. All DATA statements must appear after the specification statements. DATA statements may be interspersed with statement function statements and executable statements.
6. All statement function statements must precede all executable statements.
7. The $DO66 and $STORAGE metacommands, if present, must appear before all other statements; other metacommands may appear anywhere in the program unit.

2.3 Data Types

There are six basic data types in MS-FORTRAN:

1. integer (INTEGER*2 and INTEGER*4)
2. real (REAL*4 or REAL)
3. double precision (REAL*8 or DOUBLE PRECISION)
4. logical (LOGICAL*2 and LOGICAL*4)
5. complex (COMPLEX*8 and COMPLEX*16)
6. character

The properties of, the range of values for, and the form of constants for each type are described in the following pages; memory requirements are shown in Table 2.1.

Table 2.1
Memory Requirements

Type	Bytes	Note
LOGICAL	2 or 4	1
LOGICAL*2	2	
LOGICAL*4	4	
INTEGER	2 or 4	1
INTEGER*2	2	
INTEGER*4	4	
CHARACTER	1	2
CHARACTER*n	n	3
REAL	4	4
REAL*4	4	
REAL*8	8	5
DOUBLE PRECISION	8	
COMPLEX	8	6
COMPLEX*8	8	
COMPLEX*16	16	7

Notes for Table 2.1:

1. Either 2 or 4 bytes are used. The default is 4, but may be set explicitly to either 2 or 4 with the $STORAGE metacommand.
2. CHARACTER and CHARACTER*1 are synonymous.
3. Maximum n is 127.
4. REAL and REAL*4 are synonymous.
5. REAL*8 and DOUBLE PRECISION are synonymous.
6. COMPLEX and COMPLEX*8 are synonymous.
7. The COMPLEX*16 type is an extension to the full language standard.

Note

In some implementations, all numeric and logical data types always start on an even byte boundary.

2.3.1 Integer Data Types

The integer data type consists of a subset of the integers. An integer value is an exact representation of the corresponding integer. An integer variable occupies two or four bytes of memory, depending on the setting of the $STORAGE metacommand. A 2-byte integer, INTEGER*2, can contain any value in the range -32767 to 32767. A 4-byte integer, INTEGER*4, can contain any value in the range -2,147,483,647 to 2,147,483,647.

Integer constants consist of a sequence of one or more decimal digits or a radix specifier, followed by a string of digits in the range 0...(radix -1), where values between 10 and 35 are represented by the letters "A" through "Z", respectively.

A radix specifier consists of the character "#" optionally preceded by a string of decimal characters that represent the integer value of the radix. If the string is omitted, the radix is assumed to be 16. If the radix specifier is omitted, the radix is assumed to be 10.

Either format may be preceded by an optional arithmetic sign, plus (+) or minus (-). Integer constants must also be in range. A decimal point is not allowed in an integer constant.

Note

The range of values for both 16-bit and 32-bit integers does not include the most negative number that can be represented in 2's complement arithmetic in that number of bits. These numbers, 16#8000 and 16#80000000, are treated as "undefined" for error checking purposes.

Although the maximum 32-bit integer value is defined as 2**31-1, the compiler and runtime will read greater values which are nominally in the range up to 2**32. But these values will only be read without error if the radix is other than 10. They will be interpreted as the negative numbers with the corresponding internal representation. For example, 16 #FFFFFFFF will result in all the bits in the 32-bit integer result being set, and will have an arithmetic value of -1.

The following are examples of integer constants:

123	+1230	0
00000123	32767	-32767
-#AB05	2#010111	-36#ABZ07

An integer can be specified in MS-FORTRAN as INTEGER*2, INTEGER*4, or INTEGER. The first two specify 2-byte and 4-byte integers, respectively. INTEGER specifies either 2-byte or 4-byte integers, according to the setting of the $STORAGE metacommand (the default is four bytes).

Important

On many microprocessors, the code required to perform 16-bit arithmetic is considerably faster and smaller than the corresponding code to perform 32-bit arithmetic. Therefore, unless you set the MS-FORTRAN $STORAGE metacommand to a value of 2, programs will default to 32-bit arithmetic and may run more slowly than expected (see Section 6.2.8, "The $STORAGE Metacommand"). Setting the $STORAGE metacommand to 2 allows programs to run faster and use less code.

2.3.2 The Single Precision IEEE Real Data Type

The real data type (REAL or REAL*4) consists of a subset of the real numbers, the single precision real numbers. A single precision real value is normally an approximation of the real number desired and occupies four bytes of memory.

The range of single precision real values is approximately as follows:

8.43E-37	to	3.37E+38	(positive range)
-3.37E+38	to	-8.43E-37	(negative range)
0			(zero)

The precision is greater than six decimal digits and less than seven.

A basic real constant consists of:

1. an optional sign
2. an integer part
3. a decimal point
4. a fraction part
5. an optional exponent part

The integer and fraction parts consist of one or more decimal digits, and the decimal point is a period (.). Either the integer part or the fraction part may be omitted, but not both. Some sample real constants are:

-123.456	+123.456	123.456
-123.	+123	123.
-.456	+.456	.456

The exponent part consists of the letter "E" followed by an optionally signed integer constant of one or two digits. An exponent indicates that the value preceding it is to be multiplied by ten to the value of the exponent part's integer.

Some sample exponent parts are:

E12	E-12	E+12	E0

A real constant is either a basic real constant, a basic real constant followed by an exponent part, or an integer constant followed by an exponent part. For example:

+1.000E-2	1.E-2	1E-2
+0.01	100.0E-4	.0001E+2

All represent the same real number, one one-hundredth.

2.3.3 The Double Precision IEEE Real Data Type

The double precision real data type (REAL*8 or DOUBLE PRECISION) consists of a subset of the real numbers, the double precision real numbers. This subset is larger than the subset for the REAL (REAL*4) data type.

A double precision real value is normally an approximation of the real number desired. A double precision real value can be a positive, negative, or zero value and occupies eight bytes of memory. The range of double precision real values is approximately:

4.19D-307	to	1.67D+308	(positive range)
-1.67D+308	to	-4.19D-307	(negative range)
0			(zero)

The precision is greater than 15 decimal digits.

A double precision constant consists of:

1. an optional sign
2. an integer part
3. a decimal point
4. a fraction part
5. a required exponent part

The exponent uses "D" rather than "E" to distinguish it from single precision. The integer and fraction parts consist of one or more decimal digits, and the decimal point is a period. Either the integer part or the fraction part, but not both, may be omitted.

A double precision constant is either a basic real constant followed by an exponent part, or an integer constant followed by an exponent part. For example:

+1.123456789D-2	1.D-2	1D-2
+0.000000001D0	100.0000005D-4	.00012345D+2

The exponent part consists of the letter "D" followed by an integer constant. The integer constant may have a sign as an option. An exponent indicates that the value preceding it is to be multiplied by ten to the value of the exponent part's integer. If the exponent is zero, it must be specified as a zero.

Some sample exponent parts are:

D12	D-12	D+12	D0

Single and Double DECIMAL Floating-Point Format

Decimal floating-point numbers consist of a byte containing a sign bit and a 7-bit exponent in excess 64 notation followed by a mantissa consisting of 6 (single) and 14 (double) binary coded decimal digits packed two to a byte (if the exponent byte is zero, the number is zero.) The notation for decimal floating-point constants follows the same format as standard FORTRAN real and double precision constants.

The allowable ranges of single precision numbers are:

single	+1.0E-64 to +9.99E+62	(positive range)
	-9.99E+62 to -1.0E-64	(negative range)
	0	(zero)

Precision is exactly 6 digits.

The allowable ranges for double precision numbers are:

double	+1.0D-64 to +9.999D+62	(positive range)
	-9.99D+62 to -1.0D-64	(negative range)
	0	(zero)

Precision is exactly 14 digits.

The $DECMATH metacommand causes constants in the source file to be represented in base-10 format.

2.3.4 Complex Data Types

The COMPLEX*8 data type is an ordered pair of single precision real numbers with the second component representing an imaginary number. A COMPLEX*8 number occupies 8 bytes of memory.

A complex constant consists of an optional sign, followed by a left parenthesis, followed by two integer or real constants separated by a comma, followed by a right parenthesis.

A COMPLEX*16 data item consists of an ordered pair of double precision real numbers. COMPLEX*16 data items occupy 16 bytes of memory.

Each component (real and imaginary) of a COMPLEX*8 is a REAL*4. Each component of a COMPLEX*16 is a REAL*8.

2.3.5 Logical Data Types

The logical data type consists of the two logical values .TRUE. and .FALSE.. A logical variable occupies two or four bytes of memory, depending on the setting of the $STORAGE metacommand. The default is four bytes. The significance of a logical variable is unaffected by the $STORAGE metacommand, which is present primarily to allow compatibility with the ANSI requirement that logical, single precision real, and integer variables are all the same size.

LOGICAL*2 values occupy two bytes. The least significant (first) byte is either 0 (.FALSE.) or 1 (.TRUE.); the most significant byte is undefined. LOGICAL*4 variables occupy two words, the least significant (first) of which contains LOGICAL*2 value. The most significant word is undefined.

2.3.6 The Character Data Type

The character data type consists of a sequence of ASCII characters. The length of a character value is equal to the number of characters in the sequence. The length of a particular constant or variable is fixed, and must be between 1 and 127 characters. A character variable occupies one byte of memory for each character in the sequence.

Character variables are assigned to contiguous bytes without regard for word boundaries. However, the compiler assumes that noncharacter variables that follow character variables always start on word boundaries.

A character constant consists of a sequence of one or more characters enclosed in a pair of single right quotation marks. Blank characters are permitted in character constants and are significant. The case of alphabetic characters is significant. An apostrophe (') within a character constant is represented by two consecutive single right quotation marks with no blanks in between.

The length of a character constant is equal to the number of characters between the apostrophes. A pair of apostrophes counts as a single character. Some sample character constants are:

```
'A'
''
'Help!'
'A very long CHARACTER constant'
'O''Brien'
''''
```

The last example ('''') is a character constant that contains one apostrophe (single right quotation mark).

FORTRAN permits source lines of up to 72 columns. Shorter lines are padded with blanks to 72 columns. When a character constant extends across a line boundary, its value is as if the portion of the continuation line beginning with column 7 is appended to column 72 of the initial line.

Thus, the following FORTRAN source,

```
200     CH = 'ABC
     X DEF'
```

is equivalent to:

```
200     CH = 'ABC (58 blank spaces) ... DEF'
```

with 58 blank spaces between the C and D being equivalent to the space from C in column 15 to column 72 plus one blank in column 7 of the continuation line. Very long character constants can be represented in this manner.

2.4 Names

An MS-FORTRAN name, or identifier, consists of a sequence of alphanumeric characters (the maximum is 66 characters per line multiplied by twenty lines). The initial character must be alphabetic; subsequent characters must be alphanumeric. Blanks are skipped. Only the first six characters are significant; the rest are ignored.

A name denotes a user-defined or system-defined variable, array, or program unit. Any valid sequence of characters can be used for any FORTRAN name.

There are no reserved names as in other languages. Sequences of alphabetic characters used as keywords by the MS-FORTRAN Compiler are not confused with user-defined names. The compiler recognizes keywords by their context and in no way restricts the use of user-defined names.

Thus, a program can have, for example, an array named IF, READ, or GOTO, with no error (as long as it conforms to the rules that all arrays must obey). However, use of keywords for user-defined names often interferes with the readability of the program, so the practice should be avoided.

2.4.1 Scope of FORTRAN Names

The scope of a name is the range of statements in which that name is known, or can be referenced, within a FORTRAN program. In general, the scope of a name is either global or local, although there are several exceptions. A name can only be used in accordance with a single definition within its scope. The same name, however, can have different definitions in distinct scopes.

A name with global scope can be used in more than one program unit (a subroutine, function, or the main program) and still refer to the same entity. In fact, names with global scope can only be used in a single, consistent manner within the same program. All subroutine, function subprogram, and common block names, as well as the program name, have global scope. Therefore, there cannot be a function subprogram that has the same name as a subroutine subprogram or a common data area. Similarly, no two function subprograms in the same program can have the same name.

A name with local scope is only visible (known) within a single program unit. A name with a local scope can be used in another program unit with a different meaning or with a similar meaning, but is in no way required to have a similar meaning in a different scope. The names of variables, arrays, arguments, and statement functions all have local scope.

One exception to the scoping rules is the name given to a common data block. It is possible to refer to a globally scoped common block name in the same program unit in which an identical locally scoped name appears. This is permitted because common block names are always enclosed in slashes, such as /FROG/, and are therefore always distinguishable from ordinary names.

Another exception to the scoping rules is made for statement function arguments. The scope of statement function arguments is limited to the single statement forming that statement function. Any other use of those names within that statement function is not permitted, while any other use outside that statement function is acceptable.

2.4.2 Undeclared FORTRAN Names

As it passes over the executable statements in a program, the compiler classifies names which it encounters for the first time (i.e. those not explicitly defined) according to the context.

If the name is used as a variable, its type is inferred from the first letter of the name; I, J, K, L, M, or N by default are used to initial integers, while all others are used to initial real numbers. You can use the IMPLICIT statement to change the association of type and initial letter. (For more information, see Section 3.2.27, "The IMPLICIT Statement"). The same rules are used when the name is used in a function call to determine the type of the function return values.

When a name is used as the target of a CALL statement, it is assumed to be that of a subroutine. Similarly, a name used in a function reference is assumed to be that of a function. If a subroutine or function is part of the same compilation unit (i.e., is in the same source file), and its definition occurs before the reference to it in a CALL statement or function reference, the compiler will check that the number and type of the actual arguments in the CALL statement or function reference are consistent with those specified in the corresponding SUBROUTINE or FUNCTION statement.

2.5 Expressions

An expression is a formula for computing a value. It consists of a sequence of operands and operators. The operands may contain function invocations, variables, constants, or even other expressions. The operators specify the actions to be performed on the operands.

FORTRAN has four classes of expressions:

1. arithmetic
2. character
3. relational
4. logical

2.5.1 Arithmetic Expressions

An arithmetic expression produces a value that is of type INTEGER, REAL, DOUBLE PRECISION or COMPLEX. The simplest forms of arithmetic expressions are:

1. constants
2. variable references
3. array element references
4. function references

The value of a variable reference or array element reference must be defined before it can appear in an arithmetic expression. Moreover, the value of an integer variable must be defined with an arithmetic value, rather than a statement label value previously set in an ASSIGN statement.

Other arithmetic expressions are built up from the simple forms in the preceding list using parentheses and the arithmetic operators shown in Table 2.2.

Table 2.2
Arithmetic Operators

Operator	Operation	Precedence
**	Exponentiation	Highest
/	Division	Intermediate
*	Multiplication	Intermediate
-	Subtraction or Negation	Lowest
+	Addition or Identity	Lowest

All of the operators may be used as binary operators, which appear between their arithmetic expression operands. The plus (+) and minus (-) may also be unary, and precede their operand.

Operations of equal precedence, except exponentiation, are left-associative. Exponentiation is right-associative. Thus, each of the following expressions on the left is the same as the corresponding expression on the right:

```
A/B*C          (A/B)*C
A**B**C        A**(B**C)
```

Arithmetic expressions can be formed in the usual mathematical sense, as in most programming languages. However, FORTRAN prohibits two operators from appearing consecutively. For example, this is prohibited,

```
A**-B
```

while this is allowed:

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

Unary minus is also of lowest precedence. Thus, the expression -A**B is interpreted as -(A**B).

You may use parentheses in an expression to control associativity and the order in which operators are evaluated.

2.5.2 Integer Division

The division of two integers results in a value that is the mathematical quotient of the two values, truncated downward (i.e., toward zero). Thus, 7/3 evaluates to 2, and (-7)/3 evaluates to -2. Both 9/10 and 9/(-10) evaluate to 0 (zero).

2.5.3 Type Conversions of Arithmetic Operands

When all operands of an arithmetic expression are of the same data type, the value returned by the expression is also of that type. When the operands are of different data types, the data type of the value returned by the expression is the type of the highest-ranked operand.

The rank of an operand depends on its data type, as shown in the following list:

1. INTEGER*2 (lowest)
2. INTEGER*4
3. REAL*4
4. REAL*8
5. COMPLEX*8
6. COMPLEX*16 (highest)

For example, an operation on an INTEGER*2 and a REAL*4 element produces a value of data type REAL*4.

Special Case

Operations on operands of types REAL*8 and COMPLEX*8 yield COMPLEX*16 results, not COMPLEX*8 results as suggested in the list.

The data type of an expression is the data type of the result of the last operation performed in evaluating the expression.

The data types of operations are classified as either INTEGER*2, INTEGER*4, REAL*4, REAL*8, COMPLEX*8, or COMPLEX*16.

Integer operations are performed on integer operands only. A fraction resulting from division is truncated in integer arithmetic, not rounded. Thus, the following evaluates to zero, not one:

```
1/4 + 1/4 + 1/4 + 1/4
```

Memory allocation for the type INTEGER, without the *2 or *4 length specification, is dependent on the use of the $STORAGE metacommand. (See the note at the beginning of Section 2.3, "Data Types," and Section 6.2.12, "The $STORAGE Metacommand," for details.)

Real operations are performed on real operands or combinations of real and integer operands only. Integer operands are first converted to real data type by giving each a fractional part equal to zero. Real arithmetic is then used to evaluate the expression. In the following statement, integer division is performed on I and J, and a real multiplication on the result and X:

```
Y = (I/J)*X
```

2.5.4 Character Expressions

A character expression produces a value that is of type CHARACTER. The forms of character expressions are:

1. character constants
2. character variable references
3. character array element references
4. any character expression enclosed in parentheses
5. character function references

There are no operators that result in character expressions.

2.5.5 Relational Expressions

Relational expressions compare the values of two arithmetic or two character expressions. An arithmetic value may not be compared with a character value, unless the $NOTSTRICT metacommand has been specified. In this case, the arithmetic expression is considered to be a character expression. The result of a relational expression is of type LOGICAL. Relational expressions can use any of the operators shown in Table 2.3 to compare values.

Table 2.3
Relational Operators

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

All of the relational operators are binary operators and appear between their operands. There is no relative precedence or associativity among the relational operands since an expression of the following form violates the type rules for operands:

```
A .LT. B .NE. C
```

Relational expressions may only appear within logical expressions.

Relational expressions with arithmetic operands may have one operand of type INTEGER and one of type REAL. In this case, the integer operand is converted to type REAL before the relational expression is evaluated.

Relational expressions with character operands compare the position of their operands in the ASCII collating sequence. An operand is less than another if it appears earlier in the collating sequence. If operands of unequal length are compared, the shorter operand is considered as if it were extended to the length of the longer operand by the addition of spaces on the right.

2.5.6 Logical Expressions

A logical expression produces a value that is of type LOGICAL. The simplest forms of logical expressions are:

1. logical constants
2. logical variable references
3. logical array element references
4. logical function references
5. relational expressions

Other logical expressions are built up from the simple forms in the preceding list by using parentheses and the logical operators of Table 2.4.

Table 2.4

Logical Operators

Operator	Operation	Precedence
.NOT.	Negation	Highest
.AND.	Conjunction	Intermediate
.OR.	Inclusive disjunction	Intermediate
.EQV.	Equivalence	Lowest
.NEQV.	Nonequivalence	Lowest

The .AND., .OR., .EQV. and .NEQV. operators are binary operators and appear between their logical expression operands. The .NOT. operator is unary and precedes its operand.

Operations of equal precedence are left-associative; thus, for example,

```
A .AND. B .AND. C
```

is equivalent to:

```
(A .AND. B) .AND. C
```

As an example of the precedence rules,

```
.NOT. A .OR. B .AND. C
```

is interpreted the same as:

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

Two .NOT. operators cannot be adjacent to each other, although

```
A .AND. .NOT. B
```

is an example of an allowable expression with two adjacent operators.

As another example of the precedence rules and the use of .EQV. and .NEQV.,

```
.NOT. A .EQV. B .OR. C .NEQV. D .AND. E
```

can be interpreted as,

```
((.NOT. A) .EQV. (B .OR. C)) .NEQV. (D .AND. E)
```

Logical operators have the same meaning as in standard mathematical semantics, with .OR. being nonexclusive. For example,

```
.TRUE. .OR. .TRUE.
```

evaluates to the value:

```
.TRUE.
```

2.5.7 Precedence of Operators

When arithmetic, relational, and logical operators appear in the same expression, they abide by the following precedence guidelines:

1. Logical (lowest)
2. Relational (intermediate)
3. Arithmetic (highest)

2.5.8 Rules for Evaluating Expressions

Any variable, array element, or function that is referred to in an expression must be defined at the time the reference is made. Integer variables must be defined with an arithmetic value, rather than a statement label value as set by an ASSIGN statement.

Certain arithmetic operations, such as dividing by zero, are not mathematically meaningful and are prohibited. Other prohibited operations include raising a zero-value operand to a zero or negative power and raising a negative-value operand to a power of type REAL or DOUBLE PRECISION.

2.5.9 Array Element References

An array element reference identifies one element of an array. Its syntax is as follows:

array (*sub* [, *sub*]...)

array is the name of an array.

sub is a subscript expression, that is, an integer expression used in selecting a specific element of an array. The number of subscript expressions must match the number of dimensions in the array declarator. The value of a subscript expression must be between one and the upper limit for the dimension it represents, inclusive.

```
C EXAMPLE OF ARRAY ELEMENT REFERENCES
      DIMENSION A(3,2),B(3,4),C(4,5),D(5,6),V(10)
      EQUIVALENCE (X,V(1)), (Y,V(2))
      D(I,J) = D(I,J)/PIVOT
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
      READ(*,*) (V(N),N=1,10)
```

Chapter 3
Statements

3.1 Categories of Statements

Statements perform a number of functions, such as computing, storing the results of computations, altering the flow of control, reading and writing files, and providing information for the compiler.

FORTRAN statements fall into two broad classes: executable and nonexecutable. An executable statement causes an action to be performed. Nonexecutable statements do not in themselves cause operations to be performed. Instead, they specify, describe, or classify elements of the program, such as entry points, data, or program units.

Nonexecutable statements include the following:

1. PROGRAM, SUBROUTINE, FUNCTION , and BLOCK DATA statements
2. specification statements
3. the DATA statement
4. the FORMAT statement

The executable statements form a much larger group and may be divided into the following categories:

1. assignment statements
2. control statements
3. I/O statements

Sections 3.1.1 through 3.1.7 describe each of these types of statements, in general terms, in the order in which they are mentioned in the preceding lists.

Section 3.2, "Statement Directory," is an alphabetical listing of all statements. For each statement, the entry gives syntax and purpose, with remarks and examples as appropriate.

Chapter 4, "The I/O System," provides additional information on input and output in MS-FORTRAN.

3.1.1 PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA Statements

These statements identify the start of a program unit; all are nonexecutable. For more specific information, see the following sections: Section 3.2.4, "The BLOCK DATA Statement"; Section 3.2.20, "The FUNCTION Statement"; Section 3.2.33, "The PROGRAM Statement"; Section 3.2.38, "The Statement Function Statement"; and Section 3.2.40, "The SUBROUTINE Statement."

See also Chapter 5, "Programs, Subroutines, and Functions," for general information on program units.

3.1.2 Specification Statements

Specification statements in MS-FORTRAN are nonexecutable. They define the attributes of user-defined variable, array, and function names. Table 3.1 lists the eight specification statements, which are described in detail in Section 3.2, "Statement Directory."

Table 3.1
Specification Statements

Statement	Purpose
COMMON	Provides for sharing memory between two or more program units.
DIMENSION	Specifies that a user name is an array and defines the number of its elements.
EQUIVALENCES	Specifies that two or more variables or arrays share the same memory.
EXTERNAL	Identifies a user-defined name as an external subroutine or function.
IMPLICIT	Defines the default type for user-defined names.
INTRINSIC	Declares that a name is an intrinsic function.
PARAMETER	Equates a constant expression with an identifier (eg., name).
SAVE	Causes variables to retain their values across invocations of the procedure in which they are defined.
Type	Specifies the type of user-defined names.

Specification statements must precede all statement function statements and executable statements in a program unit. See Section 2.2.4, "Statement Definition and Order," for the rules on the order of specification statements.

3.1.3 The DATA Statement

The DATA statement assigns initial values to variables. DATA statements are optional, nonexecutable statements and may be interspersed with statement function statements and executable statements but must appear after all specification statements. (See Section 3.2.9, "The DATA Statement," for more information.)

3.1.4 The FORMAT Statement

Format specifications provide explicit editing information for the data processed by a program. Format specifications may be given in a FORMAT statement or as character constants. (See Section 3.2.19, "The FORMAT Statement," for a description of the FORMAT statement and Section 4.4, "Formatted I/O," for additional information on formatted data.)

3.1.5 Assignment Statements

Assignment statements are executable statements that assign a value to a variable or an array element. There are two basic kinds of assignment statements: computational and label. (See Section 3.2.1, "The ASSIGN Statement," and Section 3.2.2, "The Assignment Statement," respectively, for further information.)

3.1.6 Control Statements

Control statements affect the order of execution of statements in FORTRAN. The control statements in MS-FORTRAN are shown in Table 3.2, along with a brief description of the function of each. See the appropriate entries in Section 3.2, "Statement Directory," for further information on each.

Table 3.2
Control Statements

Statement	Purpose
CALL	Calls and executes a subroutine from another program unit.
CONTINUE	Used primarily as a convenient way to place statement labels, particularly as the terminal statement in a DO loop.
DO	Causes repetitive evaluation of the statements following the DO, through and including the ending statement.
ELSE	Introduces an ELSE block.
ELSEIF	Introduces an ELSEIF block.
END	Ends execution of a program unit.
ENDIF	Marks the end of a series of statements following a block IF statement.
GOTO	Transfers control elsewhere in the program, according to the kind of GOTO statement used (assigned, computed, or unconditional).
IF	Causes conditional execution of some other statement(s), depending on the evaluation of an expression and the kind of IF statement used (arithmetic, logical, or block).
PAUSE	Suspends program execution until the RETURN key is pressed.
RETURN	Returns control to the program unit that called a subroutine or function.
STOP	Terminates a program.

3.1.7 I/O Statements

I/O statements transfer data, perform auxiliary I/O operations, and position files. Table 3.3 lists the MS-FORTRAN I/O statements (each of which is described in detail in Section 3.2, "Statement Directory").

Table 3.3
I/O Statements

Statement	Purpose
BACKSPACE	Positions the file connected to the specified unit to the beginning of the previous record.
CLOSE	Disconnects the unit specified and prevents subsequent I/O from being directed to that unit.
ENDFILE	Writes an end of file record on the file connected to the specified unit.
INQUIRE	Returns values indicating the properties of a named or unit specifier.
OPEN	Associates a unit number with an external device or with a file on an external device.
READ	Transfers data from a file to the items in an *iolist*.
REWIND	Repositions a specified unit to the first record in the associated file.
WRITE	Transfers data from the items in an *iolist* to a file.

Note

Error Handling. If an error is encountered during the processing of a READ, WRITE, or INQUIRE statement, the action taken depends on the presence and definition of the ERR= and IOSTAT= specifiers. (See the statement descriptions in Section 3.2, "Statement Directory," for information about these specifiers.)

a) If neither is present, the program is terminated with an appropriate runtime error message.

b) If only ERR=*slabel2* is present, control is transferred to the designated label.

c) If only IOSTAT=*iocheck* is present, *iocheck* is set to the appropriate status value and control returns as if the statement had terminated normally.

d) If both are present, *iocheck* is set appropriately and control transfers to *slabel2*.

Any time an error is encountered in the READ statement, all the items in the *iolist* become undefined.

In addition to these I/O statements, there is an I/O intrinsic function EOF (*unit-spec*). An EOF function returns a logical value that indicates whether any data remains beyond the current position in the file associated with the given unit specifier. See Section 5.3.2, "Intrinsic Functions," for information about this function.

3.2 Statement Directory

The rest of this chapter is an alphabetical listing of all MS-FORTRAN statements, giving syntax and function, with notes and examples as necessary.

3.2.1 The ASSIGN Statement (Label Assignment)

Syntax

ASSIGN *label* TO *variable*

Purpose

Assigns the value of a format or statement label to an integer variable.

Remarks

label is a format label or statement label.

variable is an integer variable.

Execution of an ASSIGN statement sets the integer variable to the value of the label. The label can be either a format or a statement label and must appear in the same program unit as the ASSIGN statement.

When used in an assigned GOTO statement, a variable must currently have the value of a statement label. When used as a format specifier in an input/output statement, a variable must have the value of a format statement label. The only way to assign the value of a label to a variable is with the ASSIGN statement.

The value of a label is not necessarily the same as the label number. For example, the value of IVBL in the following is not necessarily 400:

```
ASSIGN 400 TO IVBL
```

Hence, the variable is undefined as an integer; it cannot be used in an arithmetic expression until it has been redefined as such (by computational assignment or a READ statement).

3.2.2 The Assignment Statement (Computational)

Syntax

variable = *expression*

Purpose

Evaluates the expression and assigns the resulting value to the variable or array element specified.

Remarks

variable is a variable or array element reference.

expression is any expression.

The type of the variable or array element and the type of expression must be compatible.

1. If the type of the right-hand side is numeric, the type of the left-hand side must be numeric, and the statement is called an arithmetic assignment statement.
2. If the type of the right-hand side is logical, the type of the left-hand side must be logical, and the statement is called a logical assignment statement.
3. If the type of the right-hand side is character, the type of the left-hand side must also be character, and the statement is called a character assignment statement. However, if you have specified the $NOTSTRICT metacommand, the type of left-hand side may be numeric, logical, or character; the statement is still called a character assignment statement.

If the types of the elements of an arithmetic assignment statement are not identical, the value of the expression is automatically converted to the type of the variable.

The conversion outcomes are given in Tables 3.4, 3.5 and 3.6. The value of the expression to be converted in the V=E assignment statement is shown after E in row one. The determining factor in the conversion, the variable type (V), is listed in column one. The outcome of the conversion (assignment) is shown in columns two and three.

Table 3.4

Conversion of Integer Values in V = E

V \ E	INTEGER*2	INTEGER*4
INTEGER*2	Assign E to V.	Assign least significant portion of E to V; most significant portion is lost.
INTEGER*4	Assign E to least significant portion of V; most significant portion is sign extended.	Assign E to V.
REAL*4	Append fraction (.0) to E and assign to V.	Append fraction (.0) to E and assign to V.
REAL*8	Append fraction (.0) to E and assign to V.	Append fraction (.0) to E and assign to V.
COMPLEX*8	Append fraction (.0) to E and assign to the real part of V.; represent imaginary part with 0.	Append fraction (.0) to E and assign to the real part of V.; represent imaginary part with 0.
COMPLEX*16	Append fraction (.0) to E and assign to the real part of V.; represent imaginary part with 0.	Append fraction (.0) to E and assign to the real part of V.; represent imaginary part with 0.

Table 3.5
Conversion of Real Values in V = E

V \ E	REAL*4	REAL*8
INTEGER*2	Truncate E to INTEGER*2 and assign to V.	Truncate E to INTEGER*2 and assign to V.
INTEGER*4	Truncate E to INTEGER*4 and assign to V.	Truncate E to INTEGER*4 and assign to V.
REAL*4	Assign E to V.	Round the least significant portion of E; assign the most significant portion to V.
REAL*8	Convert E to equivalent REAL*8 form and assign to V.	Assign E. to V.
COMPLEX*8	Assign E to real part of V; represent imaginary part with 0.	Round the least significant portion of E; assign the most significant portion to V; represent imaginary part with 0.
COMPLEX*16	Convert E to equivalent of REAL*8 form and assign real of V; represent imaginary part with 0.	Assign E to real of V; represent imaginary part with 0.

Table 3.6

Conversion of Complex Values in V = E

V \ E	COMPLEX*8	COMPLEX*16
INTEGER*2	Truncate real of E to INTEGER*2 form and assign to V; ignore imaginary part.	Truncate real of E to INTEGER*2 form and assign to V; ignore imaginary part.
INTEGER*4	Truncate real of E to INTEGER*4 form and assign to V.	Truncate real of E to INTEGER*4 form and assign to V.
REAL*4	Assign real part of E to V.	Round the least significant portion of real part of E; assign the most significant portion to V.
REAL*8	Convert real part of E to equivalent REAL*8 form and assign to V.	Assign real part of E to V.
COMPLEX*8	Assign E to V.	Round real and imaginary parts of E; assign to corresponding parts of V.
COMPLEX*16	Convert real and imaginary parts to equivalent REAL*8 form and assign to V.	Assign E to V.

For character assignments, if the length of the expression does not match the size of the variable, the expression is adjusted, in the following manner, so that it does match:

1. If the expression is shorter than the variable, the expression is padded with enough blanks on the right before the assignment takes place to make the sizes equal.
2. If the expression is longer than the variable, characters on the right are truncated to make the sizes the same.

Logical expressions of any size can be assigned to logical variables of any size without affecting the value of the expression. However, integer and real expressions may not be assigned to logical variables, nor may logical expressions be assigned to integer or real variables.

3.2.3 The BACKSPACE Statement

Syntax

BACKSPACE *unit-spec*

Purpose

Positions the file connected to the specified unit at the beginning of the preceding record.

Remarks

unit-spec is a required unit specifier; it must not be an internal unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

1. If there is no preceding record, the file position is not changed.
2. If the preceding record is the endfile record, the file is positioned before the endfile record.
3. If the file position is in the middle of the record, BACKSPACE repositions to the start of that record.
4. If the file is a binary file, the BACKSPACE repositions to the preceding byte.

Examples

```
BACKSPACE 5

BACKSPACE LUNIT
```

3.2.4 The BLOCK DATA Statement

Syntax

BLOCK DATA [*name*]

Purpose

The BLOCK DATA statement identifies a block data subprogram where the values for the variables and array elements in named common blocks are initialized.

Remarks

The BLOCK DATA statement must be the first statement in a block data subprogram.

The optional argument, *name*, is a global symbolic name for the subprogram identified by the BLOCK DATA statement. This name must be unique among the names for local variables or array elements that are defined in the subprogram which *name* labels. *name* must also be unique among the names given the main program, external procedures, common blocks, and other block data subprograms.

Note

Only one unnamed block data subprogram may appear in the executable program.

A block data subprogram may contain as many named common blocks and data initializations as desired.

The following restrictions apply to the use of block data subprograms:

1. Besides BLOCK DATA, only the COMMON, DIMENSION, PARAMETER, IMPLICIT, EQUIVALENCE, SAVE, DATA, END, and type statements may be used in the block data subprogram.
2. Named common blocks specified in block data subprograms must have unique names.
3. Only an entity defined in a named common block may be initially defined in a block data subprogram.
4. If an entity in a named common block is initially defined, all entities having storage units in the common block storage sequence must be specified even if they are not all initially defined.

3.2.5 The CALL Statement

Syntax

CALL *sname* [([*arg* [, *arg*]...])]

Purpose

Calls and executes a subroutine from another program unit.

Remarks

sname is the name of the subroutine to be called.

arg is an actual argument, which can be any of the following:

1. an alternate return specifier (**n*)
2. an expression
3. a constant (or constant expression)
4. a variable
5. an array element
6. an array
7. a subroutine name
8. an external function name
9. an intrinsic function permitted to be passed as an argument

The actual arguments in the CALL statement must agree with the corresponding formal arguments in the SUBROUTINE statement, in order, in number, and in type or kind.

The compiler will check for correspondence if the formal arguments are known. To be known, the SUBROUTINE statement that defines the formal arguments must precede the CALL state ment in the current compilation.

In addition, if the arguments are integer or logical values, agreement in size is required, according to the following rules:

1. If the formal argument is unknown, its size is determined by the $STORAGE metacommand (except as noted in rule 5 of this list). If $STORAGE is not specified, the default is $STORAGE:4.

2. If the actual argument is a constant (or constant expression), and the size of the actual argument is smaller than the size of the formal argument, a temporary variable the size of the constant will be created for the actual argument. If the actual argument is larger, an error is generated:

   95 argument type conflict

3. If the actual argument is an expression and the size of the actual argument is smaller than the size of the formal argument, then a temporary variable the size of the formal argument is created for the actual argument. If the actual argument is larger, the same error is generated as in rule 2.

4. If the actual argument is an array or a function, or if the actual argument is an array element and the formal argument an array, the compiler will not check for agreement in size.

5. If the actual argument is a variable or an array element and the formal argument is unknown, the size of the formal argument is assumed to be the same size as the size of the actual argument.

   Thus, you can call separately compiled subroutines whose formal arguments differ from the size determined by the $STORAGE metacommand in effect when the CALL is compiled. However, agreement in size is still required, and it is your responsibility to ensure this agreement.

   If the formal argument is known, then an actual argument that is a variable or an array element is treated as an expression; that is, a temporary variable for the actual argument is created if the actual argument is smaller than the formal argument. Otherwise, the same error occurs as in rule 2.

If the SUBROUTINE statement has no formal arguments, then a CALL statement referencing that subroutine must not have any actual arguments. However, a pair of parentheses may follow the subroutine name.

Execution of a CALL statement proceeds as follows:

1. All arguments that are expressions are evaluated.
2. All actual arguments are associated with their corresponding formal arguments, and the body of the specified subroutine is executed.
3. Normally, control is returned to the statement following the CALL statement upon exiting the subroutine, by executing either a RETURN statement or an END statement in that subroutine. If an alternate return in the form of RETURN *e* has been used, however, control will return to the statement specified by the *e*th alternate return specifier in the CALL statement.

For an illustration of the alternate return feature, see the following example:

```
01  CALL BAR(I,*10,J,*20,*30)
    WRITE (*,*) ' normal return'
    GO TO 40
10  WRITE (*,*) ' return to 10'
    GO TO 40
20  WRITE (*,*) ' return to 20'
    GO TO 40
30  WRITE (*,*) ' return to 30'
40  WRITE
    .
    .
    .
    SUBROUTINE BAR (I,*,J,*,*)
    IF (I.EQ.10) RETURN 1
    IF (I.EQ.20) RETURN 2
    IF (I.EQ.30) RETURN 3
    RETURN
```

(See the SUBROUTINE Statement for more details on alternate returns.)

A subroutine can be called from any program unit. Microsoft FORTRAN does not permit recursive subroutine calls. That is, a subroutine cannot call itself directly, nor can it call another subroutine that results in that subroutine being called again before it returns control to its caller.

Note

MS-FORTRAN Compiler does not detect recursive calls, even if they are direct.

Example

```
C EXAMPLE OF CALL STATEMENT
          IF (IERR .NE. 0) CALL ERROR(IERR)
          END
C
          SUBROUTINE ERROR(IERRNO)
          WRITE (*, 200) IERRNO
200       FORMAT (1X, 'ERROR', 15, 'DETECTED')
          END
```

3.2.6 The CLOSE Statement

Syntax

CLOSE (*unit-spec* [, STATUS='*status*']
[, IOSTAT=*iocheck*])

Purpose

Disconnects the unit specified and prevents subsequent I/O from being directed to that unit (unless the same unit number is re-opened, possibly associated with a different file or device). The file is discarded if the statement includes STATUS='DELETE'.

Remarks

unit-spec is a required unit specifier. It must appear as the first argument; it must not be an internal unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

status is an optional argument and may be either KEEP or DELETE. This option is a character constant and must be enclosed in single quotation marks.

If *status* is not specified, the default is KEEP, except for files opened as scratch files, which have DELETE as the default. Scratch files are always deleted upon normal program termination, and specifying STATUS='KEEP' for scratch or temporary files has no effect.

iocheck is an integer variable or integer array element that becomes defined as (1) a zero if no error or end of file conditions are encountered or (2) a processor-dependent positive integer value if an error condition is encountered or (3) a processor-dependent negative integer value if an end of file is encountered and no error condition exists.

CLOSE for unit zero has no effect, since the CLOSE operation is not meaningful for the keyboard and screen. Opened files do not have to be explicitly closed. Normal termination of an MS-FORTRAN program will close each file with its default status.

Example

This deletes an existing file:

```
C CLOSE THE FILE OPENED IN OPEN EXAMPLE,
C DISCARDING THE FILE.
      CLOSE(7,STATUS='DELETE')
```

3.2.7 The COMMON Statement

Syntax

COMMON [/[*cname*]/] *nlist* [[,] /[*cname*]/ *nlist*]...

Purpose

Provides for sharing memory between two or more program units. Such program units can manipulate the same datum without passing it as an argument.

Remarks

cname is a common block name. If a *cname* is omitted, then the blank common block is assumed.

nlist is a list of variable names, array names, and array declarators, separated by commas. Formal argument names and function names cannot appear in a COMMON statement.

In each COMMON statement, all variables and arrays appearing in each *nlist* following a common block name are declared to be in that common block. Omitting the first *cname* specifies that all elements in the first *nlist* are in the blank common block.

Any common block name can appear more than once in COMMON statements in the same program unit. All elements in all *nlists* for the same common block are allocated in that common memory area, in the order they appear in the COMMON statement(s).

The current implementation of MS-FORTRAN restricts the occurrence of noncharacter variables to even byte addresses, which may affect the association of character and noncharacter variables within a COMMON. Because of the order requirement, the compiler cannot adjust the position of variables within a COMMON to comply with the even address restriction. The compiler will generate an error message for those associations which result in a conflict.

The length of a common block is equal to the number of bytes of memory required to hold all elements in that common block. If several distinct program units refer to the same named common block, the common block must be the same length in each program unit. Blank common blocks, however, can have different lengths in different program units. The length of the blank common block is the maximum length.

Note

On some systems, other restrictions may apply. For example, the size and alignment of variables in large common blocks may be restricted.

Example

```
C EXAMPLE OF BLANK AND NAMED COMMON BLOCKS
                      PROGRAM MYPROG
                      COMMON I, J, X, K(10)
                      COMMON /MYCOM/ A(3)
                        .
                        .
                      END
                      SUBROUTINE MYSUB
                      COMMON I, J, X, K(10)
                      COMMON /MYCOM/ A(3)
                        .
                        .
                      END
```

3.2.8 The CONTINUE Statement

Syntax

CONTINUE

Purpose

Execution has no effect on the program.

Remarks

The CONTINUE statement is used primarily as a convenient point for placing a statement label, particularly as the terminal statement in a DO loop.

Example

```
C EXAMPLE OF CONTINUE STATEMENT
      DO  10 I = 1, 10
          IARRAY(I) = 0
10        CONTINUE
```

3.2.9 The DATA Statement

Syntax

DATA *nlist /clist/* [[,] *nlist /clist/*]...

Purpose

Assigns initial values to variables.

Remarks

A DATA statement is an optional, nonexecutable statement. If present, it must appear after all specification statements but may be interspersed with statement function and executable statements.

nlist is a list of variables, array elements, or array names.

clist is a list of constants, or a constant preceded by an integer constant repeat factor and an asterisk, such as:

```
5*3.14159
3*'Help'
100*0
```

A repeat factor followed by a constant is the equivalent of a list of all constants having the specified value and repeated as often as specified by the repeat constant.

There must be the same number of values in each *clist* as there are variables or array elements in the corresponding *nlist*. The appearance of an array in an *nlist* is equivalent to a list of all elements in that array in memory sequence order. Array elements must be indexed only by constant subscripts.

Normal type conversion takes place for each noncharacter element in a *clist*. Also, with the $NOTSTRICT metacommand in effect, a character element in a *clist* can correspond to a variable of any type.

The character element must have a length that is less than or equal to the length of that variable or array element. If the length of the constant is shorter, it is extended to the length of the variable by adding blank characters to the right. A single character constant cannot be used to define more than one variable or even more than one array element.

Only local variables and array elements can appear in a DATA statement. Formal arguments, variables in blank common, and function names cannot be assigned initial values with a DATA statement.

In a BLOCK DATA subprogram (exclusively) members of named common blocks may be assigned initial values with a DATA statement.

Examples

```
INTEGER N, ORDER, ALPHA

REAL COEF(4), EPS(2)

DATA N /0/, ORDER /3/

DATA  ALPHA /'A'/

DATA COEF /1.0,2*3.0,1.0/, EPS(1) /.00001/
```

3.2.10 The DIMENSION Statement

Syntax

DIMENSION *array* (*dim*) [, *array* (*dim*)]...

Purpose

Specifies that a user name is an array and defines the number of its elements.

Remarks

array is the name of an array.

dim specifies the dimensions of the array and is a list of one to seven dimension declarators separated by commas.

The number of dimensions in the array is the number of dimension declarators in the array declarator. The maximum number of dimensions is seven.

A dimension declarator can be an unsigned integer constant, a user name corresponding to a nonarray integer formal argument, a user name corresponding to a nonarray integer variable in a COMMON block in the same program unit containing the DIMENSION statement, or an asterisk.

A dimension declarator specifies the upper bound of the dimension. The lower bound is always one.

If a dimension declarator is an integer constant, then the array has the corresponding number of elements in that dimension. An array has a constant size if all of its dimensions are specified by integer constants.

If a dimension declarator is an integer formal argument or an integer variable in COMMON, then that dimension is defined to be of a size equal to the initial value of the variable upon entry to the subprogram unit at execution time. In such a case, the array is called an adjustable-size array.

If the dimension declarator is an asterisk, the array is an assumed-size array and the upper bound of that dimension is not specified.

All adjustable and assumed-size arrays must also be formal arguments to the program unit in which they appear. Furthermore, an assumed-size dimension declarator may only appear as the last dimension in an array declarator.

Array elements are stored in column-major order; the leftmost subscript changes most rapidly as the array is mapped into contiguous memory addresses.

For example, the following statements

```
INTEGER*2 A (2, 3)
DATA A /1, 2, 3, 4, 5, 6/
```

would result in the following mapping (assuming A is placed at location 1000 in memory):

Array Element	Address	Value
A (1, 1)	1000	1
A (2, 1)	1002	2
A (1, 2)	1004	3
A (2, 2)	1006	4
A (1, 3)	1008	5
A (2, 3)	100A	6

Example

```
DIMENSION A (2,3), V (10)
CALL SUBR (A,2,V)
  .
  .
SUBROUTINE SUBR (MATRIX, ROWS, VECTOR)
REAL MATRIX, VECTOR
INTEGER ROWS
DIMENSION MATRIX (ROWS,*), VECTOR (10),
+LOCAL (2,4,8)
MATRIX (1,1) = VECTOR (5)
  .
  .
END
```

3.2.11 The DO Statement

Syntax

DO *slabel* [,] *variable* = *expr1*, *expr2* [, *expr3*]

Purpose

Repeatedly evaluates the statements following the DO, through and including the statement with the label *slabel.*

Remarks

slabel is the statement label of an executable statement.

variable is an integer variable.

expr1, expr2, expr3 are integer expressions.

The label referred to must appear after the DO statement and be contained in the same program unit. The specified statement is called the terminal statement of the DO loop and must not be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSEIF, ELSE, ENDIF, RETURN, STOP, END, or DO statement. If the terminal statement is a logical IF, it may contain any executable statement except those not permitted inside a logical IF statement.

The range of a DO loop begins with the statement that follows the DO statement and includes the terminal statement of the DO loop.

The following restrictions affect the execution of a DO statement:

1. If a DO statement appears in the range of another DO loop, its range must be entirely contained within the range of the enclosing DO loop, although the loops may share a terminal statement.
2. If a DO statement appears within an IF, ELSEIF, or ELSE block, the range of the associated DO loop must be entirely contained in the particular block.

3. If a block IF statement appears within the range of a DO loop, its associated ENDIF statement must also appear within the range of that DO loop.

The DO variable may not be modified in any way by the statements within the range of the DO loop associated with it. Jumping into the range of a DO loop from outside its range is not permitted. (However, a special feature, added for compatibility with earlier versions of FORTRAN, does permit "extended range" DO loops. See Section 6.2.3, "The $DO66 Metacommand," for more information.)

In some circumstances, the value of a DO variable may overflow as a result of an increment that is performed prior to testing it against the upper bound. If this happens, your program is technically in error, but the error is not detected as such by either the compiler or the runtime. However, if the DO variable has been either explicitly or implicitly defined as INTEGER*2, and the possiblity of overflow exists, the arithmetic for the statement will be carried out in 32-bit mode with the necessary conversions and the loop will terminate.

For example:

```
      INTEGER*2 I
      DO 100 I=32760,32767
        .
        .
100   CONTINUE
```

If the DO variable is either explicitly or implicitly defined as INTEGER*4, and an overflow occurs, the value will wrap around, and the loop will not terminate.

The execution of a DO statement sets the following process in motion:

1. The expressions *expr1, expr2*, and *expr3* are evaluated. If *expr3* is not present, it is assumed that *expr3* evaluated to one.
2. The DO variable is set to the value of the expression, *expr1*.
3. The iteration count for the loop is:

```
MAX0(((expr2-expr1+expr3)/expr3),0)
```

The iteration count may be zero if either of the following is true:

a. *expr1* is greater than *expr2* and *expr3* is greater than zero

b. *expr1* is less than *expr2* and *expr3* is less than zero

However, if the $DO66 metacommand is in effect, the iteration count is at least one. See Section 6.2.2, "The $DO66 Metacommand," for more information about this feature.

4. The iteration count is tested, and, if it exceeds zero, the statements in the range of the DO loop are executed.

Following the execution of the terminal statement of a DO loop, these steps take place:

1. The value of the DO variable is incremented by the value of *expr3* that was computed when the DO statement was executed.
2. The iteration count is decremented by one.
3. The iteration count is tested, and if it exceeds zero, the statements in the range of the DO loop are executed again.

The value of the DO variable is well-defined, regardless of whether the DO loop exits because the iteration count becomes zero, or because of a transfer of control out of the DO loop.

Example

```
C EXAMPLE OF DO STATEMENTS

C INITIALIZE A 20-ELEMENT REAL ARRAY
      DIMENSION ARRAY(20)
      DO 1 I = 1, 20
1     ARRAY(I) = 0.0

C PERFORM A FUNCTION 11 TIMES
      DO 2, I = -30, -60, -3
      J = I/3
      J = -9-J
      ARRAY(J) = MYFUNC(I)
      2          CONTINUE
```

The following shows the final value of a DO variable:

```
C DISPLAY THE NUMBERS 1 TO 11 ON THE SCREEN
      DO 200 I=1,10
200   WRITE(*,'(I5)')I
      WRITE(*,'(I5)')I
```

3.2.12 The ELSE Statement

Syntax

ELSE

Purpose

Marks the beginning of an ELSE block.

Remarks

The associated ELSE block consists of all of the executable statements (possibly none) that follow the ELSE statement, up to but not including the next ENDIF statement at the same IF-level as this ELSE statement. The matching ENDIF statement must appear before any intervening ELSE or ELSEIF statements of the same IF-level. (See Section 3.2.25, "The IF THEN ELSE Statement," for a discussion of IF-levels.)

Transfer of control into an ELSE block from outside that block is not permitted.

Example

```
CHARACTER C
   .
   .
READ (*,'(A)') C
IF (C .EQ. 'A') THEN
   CALL ASUB
ELSE
   CALL OTHER
ENDIF
   .
   .
```

3.2.13 The ELSEIF Statement

Syntax

ELSEIF (*expression*) THEN

Purpose

Causes execution of a block of statements if *expression* is true.

Remarks

expression is a logical expression. If its value is true and there is at least one statement in the ELSEIF block, the next statement executed is the first statement of the ELSEIF block.

If the expression evaluates to true and the ELSEIF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF-level as the ELSEIF statement. If the expression evaluates to false, the next statement executed is the next ELSEIF, ELSE, OR ENDIF statement that has the same IF-level as the ELSEIF statement. See Section 3.2.26, "The IF THEN ELSE Statement," for a discussion of IF-levels.

The associated ELSEIF block consists of all the executable statements (possibly none) that follow, up to the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this ELSEIF statement.

Following the execution of the last statement in the ELSEIF block, the next statement to be executed is the next ENDIF statement at the same IF-level as this ELSEIF statement.

Transfer of control into an ELSEIF block from outside that block is not permitted.

Example

```
CHARACTER C
   .
   .
READ (*,'(A)') C
IF  (C .EQ. 'A') THEN
    CALL ASUB
ELSEIF (C .EQ. 'X') THEN
    CALL XSUB
ELSE
    CALL OTHER
ENDIF
```

3.2.14 The END Statement

Syntax

END

Purpose

In a subprogram, has the same effect as a RETURN statement; in the main program, terminates execution of the program. Always marks the end of the program unit in which it appears.

Remarks

The END statement must appear as the last statement in every program unit. Unlike other statements, an END statement must appear alone on an initial line, with no label. No continuation lines may follow the END statement. No other FORTRAN statement, such as the ENDIF statement, may have an initial line that appears to be an END statement.

Example

```
C EXAMPLE OF END STATEMENT
C END STATEMENT MUST BE LAST STATEMENT
C IN A PROGRAM
      PROGRAM MYPROG
      WRITE(*, '(10H HI WORLD!)')
      END
```

3.2.15 The ENDFILE Statement

Syntax

ENDFILE *unit-spec*

Purpose

Writes an end of file record as the next record of the file connected to the specified unit.

Remarks

unit-spec is a required external unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

After writing the end of file record, ENDFILE positions the file after the end of file record. This prohibits further sequential data transfer until after execution of either a BACKSPACE or REWIND statement.

An ENDFILE on a direct access file makes all records written beyond the position of the new end of file disappear.

Example

```
.
.
WRITE (6,*) X
ENDFILE 6
REWIND 6
READ (6,*) Y
.
.
```

3.2.16 The ENDIF Statement

Syntax

ENDIF

Purpose

Terminates a block IF statement. Execution of an ENDIF statement itself has no effect on the program.

Remarks

There must be a matching ENDIF statement for every block IF statement in a program unit, to identify which statements belong to a particular block IF statement. See Section 3.2.26, "The IF THEN ELSE Statement," for discussion and examples of block IFs.

Example

```
IF (I .LT. 0) THEN
    X = -1
    Y = -1
ENDIF
```

3.2.17 The EQUIVALENCE Statement

Syntax

EQUIVALENCE (*nlist*) [, (*nlist*)]...

Purpose

Specifies that two or more variables or arrays are to share the same memory.

Remarks

nlist is a list of at least two elements, separated by commas. An *nlist* may include variable names, array names, or array element names; argument names are not allowed. Subscripts must be integer constants and must be within the bounds of the array they index. No automatic type conversion occurs if the shared elements are of different types.

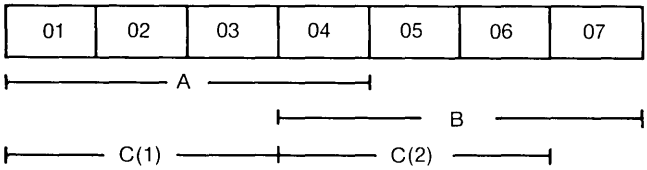
An EQUIVALENCE statement specifies that the memory sequences of the elements that appear in the list *nlist* must have the same first memory location. Two or more variables are said to be associated if they refer to the same actual memory. Thus, an EQUIVALENCE statement causes its list of variables to become associated. An array name, if present in an EQUIVALENCE statement, refers to the first element of the array.

You cannot associate character and noncharacter entities when the $STRICT metacommand is in effect ($NOTSTRICT is the default). See the odd byte boundary restriction described in number 3 in the following list.

Associated character entities may overlap, as in the following example:

```
CHARACTER A*4, B*4, C(2)*3
EQUIVALENCE (A,C(1)), (B,C(2))
```

The preceding example can be graphically illustrated as follows:



Restrictions

1. You cannot force a variable to occupy more than one distinct memory location; nor can you force two or more elements of the same array to occupy the same memory location. For example, the following statement would force R to occupy two distinct memory locations or S(1) and S(2) to occupy the same memory location:

```
C THIS IS AN ERROR
      REAL R,S(10)
      EQUIVALENCE (R,S(1)),(R,S(2))
```

2. An EQUIVALENCE statement cannot specify that consecutive array elements not be stored in sequential order. The following, for example, is not permitted:

```
C THIS IS ANOTHER ERROR
       REAL R(10),S(10)
       EQUIVALENCE (R(1),S(1)),
      +(R(5),S(6))
```

3. You cannot equivalence character and noncharacter entities so that the noncharacter entities can start on an odd byte boundary.

   For entities not in a common block, the compiler will attempt to align the noncharacter entities on word boundaries. An error message will be issued if such an alignment is not possible because of multiple equivalencing. For example, the following would result in an error, since it is not possible for both variables A and B to be word aligned:

```
CHARACTER*1 C1(10)
REAL A,B
EQUIVALENCE (A,C1(1))
EQUIVALENCE (B,C1(2))
```

For entities in a common block, since positions are fixed, it is your responsibility to assure word alignment for the noncharacter entities. An error message will be issued for any that are not word aligned.

4. An EQUIVALENCE statement cannot associate an element of type CHARACTER with a noncharacter element in a way that causes the noncharacter element to be allocated on an odd byte boundary. However, there are no boundary restrictions for equivalencing of character variables.
5. When EQUIVALENCE statements and COMMON statements are used together, several additional restrictions apply:

    a. An EQUIVALENCE statement cannot cause memory in two different common blocks to be shared.

    b. An EQUIVALENCE statement can extend a common block by adding memory elements following the common block, but not preceding the common block.

    c. Extending a named common block with an EQUIVALENCE statement must not make its length different from the length of the same named common block in other program units.

For example, the following is not permitted because it extends the common block by adding memory preceding the start of the block:

```
C THIS IS A MORE SUBTLE ERROR
                COMMON /ABCDE/ R(10)
                REAL S(10)
                EQUIVALENCE (R(1),S(10))
```

Example

```
C CORRECT USE OF EQUIVALENCE STATEMENT
      CHARACTER NAME, FIRST, MIDDLE, LAST
      DIMENSION NAME(60), FIRST(20),
     1          MIDDLE(20), LAST(20)
      EQUIVALENCE (NAME(1), FIRST(1)),
     1          (NAME(21),MIDDLE(1)),
     2          (NAME(41), LAST(1))
```

3.2.18 The EXTERNAL Statement

Syntax

EXTERNAL *name* [, *name*]...

Purpose

Identifies a user-defined name as an external subroutine or function.

Remarks

name is the name of an external subroutine or function.

Giving a name in an EXTERNAL statement declares it as an external procedure. Statement function names cannot appear in an EXTERNAL statement. If an intrinsic function name appears in an EXTERNAL statement, that name becomes the name of an external procedure, and the corresponding intrinsic function can no longer be called from that program unit. A user name can only appear once in an EXTERNAL statement in any given program unit.

In assembly language and MS-Pascal, EXTERN means that an object is defined outside the current compilation or assembly unit. This is unnecessary in MS-FORTRAN since standard FORTRAN practice assumes that any object referred to but not defined in a compilation unit is defined externally.

In FORTRAN, therefore, EXTERNAL is used primarily to specify that a particular user-defined name is a subroutine or function to be used as a procedural parameter. EXTERNAL may also indicate that a user-defined function is to replace an intrinsic function of the same name.

Examples

```
C EXAMPLE OF EXTERNAL STATEMENT
      EXTERNAL MYFUNC, MYSUB
C MYFUNC AND MYSUB ARE PARAMETERS TO CALC
      CALL CALC (MYFUNC, MYSUB)

C EXAMPLE OF A USER-DEFINED FUNCTION
C REPLACING AN INTRINSIC
      EXTERNAL SIN
      X = SIN (A,4.2,37)
```

3.2.19 The FORMAT Statement

Syntax

FORMAT (*format-spec*)

Purpose

Used in conjunction with formatted I/O statements, provides information that directs the editing of data.

Remarks

format-spec is a list format specifications, which provide explicit editing information. The *format-spec* must be enclosed in parentheses. A format specification may take one of the following forms:

[*r*] repeatable edit descriptor

nonrepeatable edit descriptor

[*r*] (*format-spec*)

The *r*, if present, is a nonzero, unsigned, integer constant called a repeat specification.

Up to three levels of nested parentheses are permitted within the outermost level of parentheses.

Edit descriptors, both repeatable and nonrepeatable, are listed in Table 3.7 and described in more detail in Section 4.4.2, "Edit Descriptors."

You may omit the comma between two list items if the resulting format specification is not ambiguous; for example, after a P edit descriptor or before or after the slash (/) edit descriptor.

FORMAT statements must be labeled and, like all nonexecutable statements, cannot be the target of a branching operation.

Table 3.7
Edit Descriptors

Repeatable	Nonrepeatable
I*w*	*'xxx'* (character constants)
G*w.d*	*n*H*xxx* (character constants)
G*w.d*E*e*	*n*X (positional editing)
F*w.d*	/ (terminate record)
E*w.d*	\ (don't terminate record)
E*w.d*E*e*	*k*P (scale factor)
D*w.d*	BN (blanks as blanks or ignored)
L*w*	BZ (blanks as zeros)
A[*w*]	T*c* (positional editing)
	TR*c* (positional editing)
	TL*c* (positional editing)
	: (format scan terminator)
	SP (optional plus character control)
	SS (optional plus character control)
	S (optional plus character control)

Notes for Table 3.7

A) For the repeatable edit descriptors:

1. A, D, E, F, G, I, and L indicate the manner of editing.
2. (*w*) and (*e*) are nonzero, unsigned, integer constants.
3. (*d*) is an unsigned integer constant.

B) For the nonrepeatable edit descriptors:

1. ('), H, X, (/), (\), P, BN, BZ, T, TL, TR, S, SS, SP, and (:) indicate the manner of editing.
2. (*x*) is any ASCII character.
3. (*n*) is a nonzero, unsigned, integer constant.
4. (*k*) is an optionally signed integer constant.
5. (*c*) is an unsigned integer constant.

Note

Invalid format strings cause warning messages.

See Section 4.4, "Formatted I/O," for further information on edit descriptors and formatted I/O.

3.2.20 The FUNCTION Statement (External)

Syntax

[*type*] FUNCTION *fname* ([*farg* [, *farg*]...])

Purpose

Identifies a program unit as a function and supplies its type, name, and optional formal parameter(s).

Remarks

type is one of the following:

INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
LOGICAL
LOGICAL*2
LOGICAL*4
CHARACTER
CHARACTER*n
COMPLEX
COMPLEX*8
COMPLEX*16

fname is the user-defined name of the function.

farg is a formal argument name.

The function name is global, but it is also local to the function it names. If *type* is omitted from the FUNCTION statement, the function's type is determined by default and by any subsequent IMPLICIT or type statements that would determine the type of an ordinary variable.

If *type* is present, then the function name cannot appear in any additional type statements.

Note

CHARACTER-typed functions may not be declared with an asterisk (*) as a length specifier. For example, the FUNCTION statement,

```
CHARACTER *(*) F(X)
```

is not allowed.

If a function is CHARACTER-typed, then *n* may be specified in the following range ($1 <= n <= 127$).

The list of argument names defines the number and, with any subsequent IMPLICIT, EXTERNAL, type, or DIMENSION statements, the type of arguments to that function. Neither argument names nor the function name can appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

The function name must appear as a variable in the program unit that defines the function. Every execution of that function must assign a value to that variable. The final value of this variable, upon execution of a RETURN or an END statement, defines the value of the function.

Note

Alternate return specifiers are not allowed in FUNCTION statements.

After being defined, the value of this variable can be referenced in an expression, like any other variable. An external function may return values in addition to the value of the function by assignment to one or more of its formal arguments.

A function can be called from any program unit. However, FORTRAN does not allow recursive function calls, which means that a function cannot call itself directly, nor can it call another function if such a call results in that function being called again before it returns control to its caller. However, recursive calls are not detected by the compiler, even if they are direct.

Example

```
C EXAMPLE OF A FUNCTION REFERENCE
C GETNO IS A FUNCTION THAT READS A
C NUMBER FROM A FILE
          I=2
10        IF (GETNO(I) .EQ. 0.0) GO TO 10
          STOP
          END

C
          FUNCTION GETNO(NOUNIT)
          READ(NOUNIT, '(F10.5)') R
          GETNO = R
          RETURN
          END
```

3.2.21 The GOTO Statement (Assigned GOTO)

Syntax

GOTO *name* [[,] (*slabel* [, *slabel*]...)]

Purpose

Causes the statement labeled by the label last assigned to *name* to be the next statement executed.

Remarks

name is an integer variable name.

slabel is a statement label of an executable statement in the same program unit as the assigned GOTO statement.

The same statement label may appear repeatedly in the list of labels. When the assigned GOTO statement is executed, *name* must have been assigned the label of an executable statement found in the same program unit as the assigned GOTO statement.

Including the optional list of labels and selecting the $DEBUG metacommand results in a runtime error if the label last assigned to *name* is not among those listed. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted.

A special feature, extended range DO loops, does permit jumping into a DO block. See Section 6.2.3, "The $DO66 Metacommand," for more information about this feature.

Example

```
C EXAMPLE OF ASSIGNED GOTO
      ASSIGN 10 TO I
      GOTO I
10    CONTINUE
```

3.2.22 The GOTO Statement (Computed GOTO)

Syntax

GOTO (*slabel* [, *slabel*]..) [,] *i*

Purpose

Transfers control to the statement labeled by the *i*th label in the list.

Remarks

slabel is the statement label of an executable statement from the same program unit as the computed GOTO statement. The same statement label may be repeated in the list of labels.

i is an integer expression.

If there are n labels in the list of labels and *i* is out of range, the computed GOTO statement serves as a CONTINUE statement. *i* would be out of range in either of the following cases:

$i < 1$

$i > \text{n}$

Otherwise, the next statement executed is the one labeled by the *i*th label in the list of labels.

Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. A special feature, extended range DO loops, does permit jumping into a DO block. See Section 6.2.3, "The $DO66 Metacommand," for more information.

Example

```
C EXAMPLE OF COMPUTED GOTO
          I = 1
          GOTO (10, 20) I
            .
            .
10        CONTINUE
            .
            .
20        CONTINUE
```

3.2.23 The GOTO Statement (Unconditional GOTO)

Syntax

GOTO *slabel*

Purpose

Transfers control to the statement labeled *slabel.*

Remarks

slabel is the statement label of an executable statement in the same program unit as the GOTO statement.

Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. A special feature, extended range DO loops, does permit jumping into a DO block. See Section 6.2.3, "The $DO66 Metacommand," for more information about this feature.

Example

```
C EXAMPLE OF UNCONDITIONAL GOTO
                      GOTO 4022
                       .
                       .
      4022            CONTINUE
```

3.2.24 The IF Statement (Arithmetic IF)

Syntax

IF (*expression*) *slabel1, slabel2, slabel3*

Purpose

Evaluates the expression and transfers control to the statement labeled by one of the specified labels, according to the result of the expression.

Remarks

expression is an integer or real expression.

slabel1, slabel2, and slabel3 are statement labels of executable statements in the same program unit as the arithmetic IF statement.

The same statement label may appear more than once among the three labels. The first label is selected if the value of the expression is less than zero, the second label if the value equals zero, and the third label if the value is greater than zero. The next statement executed is the statement labeled by the selected label.

Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. A special feature, extended range DO loops, does permit jumping into a DO block. See Section 6.2.3, "The $DO66 Metacommand," for more information about this feature.

Example

```
C EXAMPLE OF ARITHMETIC IF
          I = 0
          IF (I) 10, 20, 30
10        CONTINUE
            .
            .
20        CONTINUE
            .
            .
30        CONTINUE
```

3.2.25 The IF Statement (Logical IF)

Syntax

IF (*expression*) *statement*

Purpose

Evaluates the logical expression and, if the value of that expression is .TRUE., executes the statement given. If the expression evaluates to .FALSE., the statement is not executed and execution continues as if a CONTINUE statement were encountered.

Remarks

expression is a logical expression.

statement is any executable statement except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement.

Example

```
C EXAMPLE OF LOGICAL IF
          IF (I .EQ. 0) J = 2
          IF (X .GT. 2.3) GOTO 100
            .
            .
100       CONTINUE
```

3.2.26 The IF THEN ELSE Statement (Block IF)

Syntax

IF (*expression*) THEN

Purpose

Evaluates the expression and, if the expression evaluates to .TRUE., begins executing statements in the IF block. If the expression evaluates to .FALSE., control transfers to the next ELSE, ELSEIF, or ENDIF statement at the same IF-level.

Remarks

expression is a logical expression.

The associated IF block consists of all the executable statements (possibly none) that appear following the statement, up to but not including the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this block IF statement.

After execution of the last statement in the IF block, the next statement executed is the next ENDIF statement at the same IF-level as this block IF statement. If the expression in this block IF statement evaluates to .TRUE., and the IF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF-level as the block IF statement. If the expression evaluates to .FALSE., the next statement executed is the next ELSEIF, ELSE, or ENDIF statement at the same IF-level as the block IF statement.

Transfer of control into an IF block from outside that block is not permitted.

IF-Levels:

The concept of an IF-level in block IF and associated statements is described as follows. For any statement, its IF-level is n1 minus n2, where:

1. n1 is the number of block IF statements from the beginning of the program unit in which the statement occurs, up to and including that statement.
2. n2 is the number of ENDIF statements from the beginning of the program unit, up to, but not including, that statement.

   The IF-level of every statement must be greater than or equal to zero and the IF-level of every block IF, ELSEIF, ELSE, and ENDIF must be greater than zero. Finally, the IF-level of every END statement must be zero. The IF-level defines the nesting rules for the block IF and associated statements and defines the extent of IF, ELSEIF, and ELSE blocks.

Example 1

Simple block IF that skips a group of statements if the expression is false:

```
IF(I.LT.10)THEN
  .          Some statements executed
  .          only if I.LT.10
ENDIF
```

Example 2

Block IF with ELSEIF statements:

```
IF(J.GT.1000)THEN
  .          Some statements executed
  .          only if J.GT.1000
ELSEIF(J.GT.100)THEN
  .          Some statements executed
  .          only if J.GT.100 and J.LE.1000
ELSEIF(J.GT.10)THEN
  .          Some statements executed
  .          only if J.GT.10 and J.LE.100
ELSE
  .          Some statements executed
  .          only if J.LE.10
ENDIF
```

Example 3

Nesting of constructs and use of an ELSE statement following a block IF without intervening ELSEIF statements:

```
IF(I.LT.100)THEN
  .          Some statements executed
  .          only if I.LT.100
      IF(J.LT.10)THEN
        .    Some statements executed
        .    only if I.LT.100 and J.LT.10
      ENDIF
  .          Some statements executed
  .          only if I.LT.100
ELSE
  .          Some statements executed
  .          only if I.GE.100
      IF(J.LT.10)THEN
        .    Some statements executed
        .    only if I.GE.100 and J.LT.10
      ENDIF
  .          Some statements executed
  .          only if I.GE.100
ENDIF
```

3.2.27 The IMPLICIT Statement

Syntax

IMPLICIT *type* (*a* [, *a*]...) [*type* (*a* [, *a*]...)...]

Purpose

Defines the default type for user-declared names.

Remarks

type is one of the following types:

INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
COMPLEX
COMPLEX*8
COMPLEX*16
LOGICAL
LOGICAL*2
LOGICAL*4
CHARACTER
CHARACTER**n*

a is either a single letter or a range of letters. A range of letters is indicated by the first and last letters in the range, separated by a minus sign. The letters for a range must be in alphabetical order.

n (as in CHARACTER**n*) may be in the following range (1 <= *n* <= 127).

An IMPLICIT statement defines the type and size for all user-defined names that begin with the letter or letters given. An IMPLICIT statement applies only to the program unit in which it appears and does not change the type of any intrinsic function.

IMPLICIT types for any specific user name can be overridden or confirmed if that name is given in a subsequent type statement. An explicit type in a FUNCTION statement also takes priority over an IMPLICIT statement. If the type in question is a character type, the length is also overridden by a later type definition.

A program unit can have more than one IMPLICIT statement. However, all IMPLICIT statements must precede all other specification statements in that program unit. The same letter cannot be defined more than once in an IMPLICIT statement in the same program unit.

Example

```
C EXAMPLE OF IMPLICIT STATEMENT
      IMPLICIT INTEGER (A - B)
      IMPLICIT CHARACTER*10 (N)
      AGE = 10
      NAME = 'PAUL'
```

3.2.28 The INQUIRE Statement

Syntax

INQUIRE (UNIT=*unit-spec* [, specifier=target]...)
or INQUIRE (FILE=*filename* [, specifier=target]...)

Purpose

The INQUIRE statement is used to examine the properties of a connected unit or a named file.

Remarks

The INQUIRE statement determines the attributes of a file and assigns the values of the attributes named by the specifiers to the corresponding targets. A target must be a variable or array element name.

The INQUIRE statement may be executed at any time. The values it returns are those that are current at the time of the call.

If you inquire by unit, the unit specifier, UNIT=, must be in the list but FILE= may not be in the list. If you inquire by file, the file specifier, FILE=, must be in the list but UNIT= is not allowed.

The discussion of inquiry specifiers that follows summarizes the specifiers that MS-FORTRAN supports.

UNIT=*unit-spec* must be the first specifier in an inquire-by-unit. *unit-spec* is either:

a) an integer (external unit)

b) an asterisk (*) identifying a processor-determined unit that is preconnected for formatted sequential access (external unit)

FILE=*filename* gives a name for the file in an inquiry by file and must be the first specifier in an inquire-by-file. The filename must be a character variable or array element.

ERR=*slabel2*. *slabel2* is the statement label of an executable statement that appears in the same program unit as the error specifier. If an error occurs, control will be transferred to this label.

EXIST=*logical-exist*. *logical-exist* is a logical variable or logical array element. Execution of INQUIRE by FILE= sets the variable .TRUE. if the specified file exists and .FALSE. if the specified file does not exist. Execution of INQUIRE by UNIT= sets the variable .TRUE. if the specified unit exists, and .FALSE. otherwise.

NAMED=*logical-named*. *logical-named* is a logical variable or a logical array element. Execution of INQUIRE by UNIT= sets the variable .TRUE. if the file was opened by name and .FALSE., otherwise. If the value of *logical-named* is .FALSE., then the file connected to the unit is a temporary file. This is one way of distinguishing temporary files from other files.

A unit number is not named if it is not open or it is open to a scratch file.

IOSTAT=*iocheck*. *iocheck* is an integer variable or integer array element that becomes defined as (1) a zero if no error or end of file conditions are encountered or (2) a processor-dependent positive integer value if an error condition is encountered or (3) a processor-dependent negative integer value if an end of file is encountered and no error condition exists.

OPENED=*logical-opened*. *logical-opened* is a logical variable or logical array element. In an inquire-by-file, it is set to .TRUE. if the named file is currently connected to any unit. Otherwise, it is set to .FALSE. In an inquire-by-unit, it is set to .TRUE. if any file is open on the given unit, and .FALSE. otherwise.

NUMBER=*num*. *num* is an integer variable or integer array element. *num* becomes undefined if no unit is connected to the file. Otherwise, in an inquire-by-file, *num* is set to the unit number connected to the file.

NAME=*filename*. *filename* is a character variable or character array element. In an inquire-by-unit, *filename* is set to the name of the file. *filename* becomes undefined if the file doesn't have a name, or if there is no file connected to the unit.

ACCESS=*type-access. type-access* is a character variable or character array element that is set to 'SEQUENTIAL' if a file is connected for sequential access or 'DIRECT' if a file is connected for direct access. If no file is connected to the given unit, *type-access* becomes undefined.

SEQUENTIAL=*logical-sequential. logical-sequential* is a character variable or character array element that is set to 'YES' if sequential is among the set of allowable access modes for the connected file, or 'NO' or 'UNKNOWN' otherwise.

DIRECT=*logical-direct. logical-direct* is character variable or character array element that is set to 'YES' if direct is among the set of allowable access modes for the connected file, or 'NO' or 'UNKNOWN' otherwise.

FORM=*format-connection. format-connection* is a character variable or character array element that is set to 'FORMATTED' if the file is connected for formatted I/O or 'UNFORMATTED' otherwise.

FORMATTED=*logical-formatted. logical-formatted* is a character variable or character array element that is set to 'YES' if formatted is among the set of allowable forms of the file; 'NO' or 'UNKNOWN' otherwise.

UNFORMATTED=*logical-unformatted. logical-unformatted* is a character variable or character array element that is set to 'YES' if unformatted is among the set of allowable forms of the file; 'NO' or 'UNKNOWN' otherwise.

RECL=*rec-length. rec-length* is an integer variable or array element name that specifies the length (in bytes) of each record in a file that is connected for direct access. If the file is connected for unformatted I/O the value will be in processor-dependent units.

NEXTREC=*nextrec-num. nextrec-num* is an integer variable or integer array element that is assigned the record number of the next record in a file that is connected for direct access. The first record in such a file has record number 1.

BLANK=*blank. blank* is a character variable or character array element that is set to 'NULL' if the BN edit descriptor is in effect or 'ZERO' if BZ is in effect.

3.2.29 The INTRINSIC Statement

Syntax

INTRINSIC *name1* [, *name2*]...

Purpose

Declares that a name is an intrinsic function.

Remarks

name is an intrinsic function name.

Each user name may appear only once in an INTRINSIC statement. A name that appears in an INTRINSIC statement cannot appear in an EXTERNAL statement. All names used in an INTRINSIC statement must be system-defined INTRINSIC functions. For a list of these functions, see Table 5.1 in Chapter 5, "Programs, Subroutines, and Functions."

You must specify the name of an intrinsic function in an INTRINSIC statement if you wish to pass it as an argument.

Example

```
C EXAMPLE OF INTRINSIC STATEMENT
      INTRINSIC SIN, COS
C SIN AND COS ARE ARGUMENTS TO CALC2
      X = CALC2 (SIN, COS)
```

3.2.30 The OPEN Statement

Syntax

OPEN (*unit-spec* [, FILE=*fname*]
[, STATUS='status'] [, ACCESS='access']
[, FORM='format'] [, IOSTAT=*iocheck*]
[, RECL=*rec-length*])

Purpose

Associates a unit number with an external device or file on an external device.

Remarks

unit-spec is a required unit specifier. It must appear as the first argument; it must not be an internal unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

fname is a character expression. This optional argument, if present, must appear as the second argument. If the argument is omitted, the compiler creates a temporary scratch file with a name unique to the unit. The scratch file is deleted when it is either explicitly closed or the program terminates normally.

If the filename specified is blank (FILE=' '), the user will be prompted for a filename at runtime. If opened with STATUS ='OLD', the file itself must exist.

All arguments after *fname* are optional and can appear in any order. Except for RECL= and IOSTAT=, these options are character constants with optional trailing blanks and must be enclosed in single quotation marks.

'status' is OLD (the default) or NEW. OLD is for reading or writing existing files; NEW is for writing new files.

'access' is SEQUENTIAL (the default) or DIRECT.

'format' is FORMATTED, UNFORMATTED, or BINARY. If access is SEQUENTIAL, the default is FORMATTED; if access is DIRECT, the default is UNFORMATTED.

iocheck is an integer variable or integer array element that becomes defined as (1) a zero if no error or end of file conditions are encountered or (2) a processor-dependent positive integer value if an error condition is encountered or (3) a processor-dependent negative integer value if an end of file is encountered and no error condition exists.

rec-length (record length) is an integer expression that specifies the length of each record in bytes. This argument is applicable only for DIRECT access files, for which it is required.

Associating unit zero to a file has no effect: Unit zero is permanently connected to the keyboard and screen.

Example 1

```
C PROMPT USER FOR A FILE NAME.
      WRITE(*,'(A\)')'Output file name?'
C PRESUME THAT FNAME IS SPECIFIED TO BE
C CHARACTER*64.
C READ THE FILE NAME FROM THE KEYBOARD.
      READ(*,'(A)') FNAME
C OPEN THE FILE AS FORMATTED SEQUENTIAL
C AS UNIT 7.
C NOTE THAT THE ACCESS SPECIFIED WAS
C UNNECESSARY SINCE IT IS THE DEFAULT.
C FORMATTED IS ALSO THE DEFAULT.
      OPEN(7,FILE=FNAME,ACCESS='SEQUENTIAL',
     +STATUS='NEW')
```

Example 2

```
C OPEN AN EXISTING FILE CREATED BY EDITOR
C CALLED DATA3.TXT AS UNIT 3.
      OPEN(3,FILE='DATA3.TXT')
```

3.2.31 The PARAMETER Statement

Syntax

PARAMETER (*p*=*e* [, *p*=*e*]....)

Purpose

The PARAMETER statement is used to give a constant a symbolic name.

Remarks

The symbolic name (*p*) must match the type of the expression (*e*). For example, if (*e*) is an arithmetic constant expression, then (*p*) must be of that type. If (*e*) is a logical or character constant expression, (*p*) must be of that type. To use a symbolic name in subsequent expressions, it must be defined in the same or in a previous PARAMETER statement in the same program unit.

If a constant's symbolic name is not of the default value of either integer or real and if the symbolic name is not of the default length, a type-statement or IMPLICIT statement must declare it before it appears in the source.

A symbolic name cannot be used in format specifications and in some other contexts for example, in a COMPLEX constant.

If (e) is not of type INTEGER, it must be a constant.

Example 1

```
PARAMETER (NBLOCKS = 10)
```

Example 2

```
REAL MASS
PARAMETER (MASS = 47.3)
```

Example 3

```
IMPLICIT REAL (L-M)
PARAMETER (LOAD = 10.0, MASS = 32.2)
```

3.2.32 The PAUSE Statement

Syntax

PAUSE [*n*]

Purpose

Suspends program execution until the RETURN key is pressed.

Remarks

n is either a character constant or a string of not more than five digits.

The PAUSE statement suspends execution of the program, pending an indication that it is to continue. The argument *n*, if present, is displayed on the screen as a prompt requesting input from the keyboard. If *n* is not present, the following message is displayed on the screen:

```
PAUSE. Please press return to continue.
```

After you press the RETURN key, program execution resumes as if a CONTINUE statement were executed.

Example

```
C EXAMPLE OF A PAUSE STATEMENT
      IF (IWARN .EQ. 0) GOTO 300
      PAUSE 'WARNING: IWARN IS NONZERO'
300   CONTINUE
```

3.2.33 The PROGRAM Statement

Syntax

PROGRAM *program-name*

Purpose

Identifies the program unit as a main program and gives it a name.

Remarks

program-name is the name you have given to your main program. The program name is a global name. Therefore, it cannot be the same as that of another external procedure or common block. (It is also a local name to the main program and must not conflict with any local name in the main program.) The PROGRAM statement may only appear as the first statement of a main program.

If the main program does not have a program statement, it will be assigned the name MAIN. The name MAIN then cannot be used to name any other entity.

Example

```
PROGRAM GAUSS
REAL COEF (10,10), CONST (10)
  .
  .
END
```

3.2.34 The READ Statement

Syntax

READ (*unit-spec* [, *format-spec*]
[, IOSTAT=*iocheck*] [, REC=*rec-num*]
[, END=*slabel1*] [, ERR=*slabel2*]) *iolist*

Purpose

Transfers data from the file associated with *unit-spec* to the items in the *iolist*, assuming that no end of file or error occurs.

Remarks

If the READ is internal, the character variable or character array element specified by *unit-spec* is the source of the input; if the READ is not internal, the source of the input is the external unit.

unit-spec is a required unit specifier, which must appear as the first argument.

format-spec is a format specifier. It is required for formatted read as the second argument; it must not appear for unformatted read.

Other arguments, if present, can appear in any order.

iocheck is an integer variable or integer array element that becomes defined as (1) a zero if no error or end of file conditions are encountered or (2) a processor-dependent positive integer value if an error condition is encountered or (3) a processor-dependent negative integer value if an end of file is encountered and no error condition exists. (See Section 3.1.7, "I/O Statements," for details on error handling).

rec-num is a record number, specified for direct access files only; if *rec-num* is given for other than direct files, an error results. The record number is a positive integer expression and positions to the record number *rec-num* (the first record in the file has record number 1) before the transfer of data begins. If this argument is omitted for a direct access file, reading continues sequentially from the current position in the file.

slabel1 is an optional statement label in the same program unit as the READ statement. If this argument is omitted, reading past the end of the file results in a runtime error. If it is present, encountering an end of file condition transfers control to the executable statement specified.

slabel2 is an optional statement label in the same program unit as the READ statement. If this argument is omitted, I/O errors result in runtime errors. If it is present, I/O errors transfer control to the executable statement specified. (See Section 3.1.7, "I/O Statements," for details on error handling).

iolist specifies the entities into which values are transferred from the file. An *iolist* may be empty, but ordinarily consists of input entities and implied DO lists, separated by commas.

See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

If the file has not been opened by an OPEN statement, an implicit OPEN operation is performed. This operation is equivalent to the following statement:

```
OPEN (unit-spec,FILE=' ',STATUS='OLD',
+ACCESS='SEQUENTIAL',FORM='format')
```

format is 'FORMATTED' if the READ statement is formatted and 'UNFORMATTED if the READ statement is unformatted. See Section 3.2.28, "The OPEN Statement," for a description of the effect of the FILE= parameter.

Example

```
C SET UP A TWO DIMENSIONAL ARRAY.
      DIMENSION IA(10,20)

C READ IN THE BOUNDS FOR THE ARRAY.
C THESE BOUNDS SHOULD BE LESS THAN OR
C EQUAL TO 10 AND 20 RESPECTIVELY.
C THEN READ IN THE ARRAY IN NESTED
C IMPLIED DO LISTS WITH INPUT FORMAT OF
C 8 COLUMNS OF WIDTH 5 EACH.
      READ (3,990)IL,JL,((IA(I,J),J=1,JL),
     +I=1,IL)
990   FORMAT (215/,(8I5))
```

3.2.35 The RETURN Statement

Syntax

RETURN [*ordinal*]

Purpose

Returns control to the calling program unit and, where the actual arguments of the CALL statement contain alternate return specifiers, can return control to a specific statement.

Remarks

RETURN can only appear in a function or subroutine.

Execution of a RETURN statement terminates execution of the enclosing subroutine or function. If the RETURN statement is in a function, the function's value is equal to the current value of the variable with the same name as the function.

Execution of an END statement in a function or subroutine is equivalent to execution of a RETURN statement. Thus, either a RETURN or an END statement, but not both, is required to terminate a function or subroutine.

[*ordinal*] defines an ordinal position for an alternate return label in the formal argument list for the subroutine. (See the SUBROUTINE statement).

Example

```
C EXAMPLE OF RETURN STATEMENT
C THIS SUBROUTINE LOOPS UNTIL
C YOU TYPE "Y"
          SUBROUTINE LOOP
          CHARACTER IN
C
10        READ(*,'(A1)') IN
          IF (IN .EQ. 'Y') RETURN
          GOTO 10
C RETURN IMPLIED
          END
```

The following code fragment is an example of the alternate return feature:

```
01        CALL BAR(I,*10,J,*20,*30)
          WRITE (*,*) ' normal return'
          GO TO 40
          WRITE
          .
          .
          .
          SUBROUTINE BAR (I,*,J,*,*)
          IF (I.EQ.10) RETURN 1
          IF (I.EQ.20) RETURN 2
          IF (I.EQ.30) RETURN 3
          RETURN
```

In this example of a subroutine with alternate return labels following the RETURN statement, RETURN 2 specifies a return to the second alternate return label in the list, RETURN 3 to the third and so on.

3.2.36 The REWIND Statement

Syntax

REWIND *unit-spec*

Purpose

Repositions to its initial point the file associated with the specified unit.

Remarks

unit-spec is a required external unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

Example

```
INTEGER A(80)
  .
  .
WRITE (7,'(80I1)') A
  .
  .
REWIND 7
  .
  .
READ (7, '(80I1)') A
```

3.2.37 The SAVE Statement

Syntax

SAVE *cname1* [, *cname2*]...

Purpose

Causes variables to retain their values across invocations of the procedure in which they are defined.

Remarks

cname is the name of a common block (enclosed in slashes), a variable or an array. After being saved, the named variables and all variables in the named common block have defined values if the current procedure is subsequently re-entered.

Example

```
C EXAMPLE OF SAVE STATEMENT
      SAVE /MYCOM/, MYVAR
```

3.2.38 The Statement Function Statement

Syntax

fname ([*farg* [, *farg*]..]) = *expr*

Purpose

Defines a function in one statement.

Remarks

fname is the name of the statement function.

farg is a formal argument name.

expr is any expression.

The statement function statement is similar in form to the assignment statement. A statement function statement can only appear after the specification statements and before any executable statements in the program unit in which it appears.

A statement function is not an executable statement, since it is not executed in order as the first statement in its particular program unit. Rather, the body of a statement function serves to define the meaning of the statement function. Like any other function, a statement function is executed by a function reference in an expression.

The type of the expression must be assignment compatible with the type of the statement function name. The list of formal argument names serves to define the number and type of arguments to the statement function. The scope of formal argument names is the statement function. Therefore, formal argument names can be re-used as other user-defined names in the rest of the program unit enclosing the statement function definition.

The name of the statement function, however, is local to the enclosing program unit; it must not be used otherwise, except as the name of a common block or as the name of a formal argument to another statement function. In the latter case the type of all such uses must be the same.

If a formal argument name is the same as another local name, then a reference to that name within the statement function defining it always refers to the formal argument, never to the other usage.

Within the expression *expr*, references to variables, formal arguments, other functions, array elements, and constants are permitted. Statement function references, however, must refer to statement functions defined prior to the statement function in which they appear. Statement functions cannot be called recursively, either directly or indirectly.

A statement function can only be referenced in the program unit in which it is defined. The name of a statement function cannot appear in any specification statement, except in a type statement (which may not define that name as an array) and in a COMMON statement (as the name of a common block). A statement function cannot be of type CHARACTER.

Example

```
C EXAMPLE OF STATEMENT FUNCTION STATEMENT
      DIMENSION X(10)
      ADD(A, B) = A + B
C
      DO 1, I=1, 10
      X(I) = ADD(Y, Z)
1     CONTINUE
```

3.2.39 The STOP Statement

Syntax

STOP [*n*]

Purpose

Terminates the program.

Remarks

n is either a character constant or a string of not more than five digits.

The argument, *n*, if present, is displayed on the screen when the program terminates. If *n* is not present, the following message is displayed:

```
STOP - Program terminated.
```

Example

```
C EXAMPLE OF STOP STATEMENT
      IF (IERROR .EQ. 0) GOTO 200
      STOP 'ERROR DETECTED'
200   CONTINUE
```

3.2.40 The SUBROUTINE Statement

Syntax

SUBROUTINE *subroutine-name* [([*farg* [, *farg*]...])]

Purpose

Identifies a program unit as a subroutine, gives it a name, and identifies the formal arguments to that subroutine. These arguments may include alternate return labels (*).

An alternate return label identifies an ordinal position [*e*] among the other alternate return labels in the formal argument list.

For example:

```
CALL BAR(I,*10,J,*20,*30)
.
.
.
SUBROUTINE BAR (I,*,J,*,*)
IF (I.EQ.10) RETURN 1
IF (I.EQ.20) RETURN 2
IF (I.EQ.30) RETURN 3
RETURN
```

RETURN 2 references the second alternate return label(*) in SUBROUTINE BAR. The second alternate return label serves as the symbol for the actual argument *20 (with its alternate return specifier) in the CALL statement.

Remarks

subroutine-name is the user-defined, global, external name of the subroutine.

farg is the user-defined name of a formal argument, also known as a dummy argument. The formal argument may include the alternate return label (*).

A subroutine begins with a SUBROUTINE statement and ends with the next following END statement. It can contain any kind of statement other than a PROGRAM statement, BLOCK DATA statement, SUBROUTINE statement, or a FUNCTION statement.

The list of argument names defines the number and, with any subsequent IMPLICIT, EXTERNAL, type, or DIMENSION statements, the type of arguments to that subroutine. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

The actual arguments in the CALL statement that reference a subroutine must agree with the corresponding formal arguments in the SUBROUTINE statement, in order, in number, and in type or kind.

The compiler will check for correspondence if the formal arguments are known. To be known, the SUBROUTINE statement that defines the formal arguments must precede the CALL statement in the current compilation. Rules for the correspondence of formal and actual arguments are described in Section 3.2.5, "The CALL Statement."

Example

```
        SUBROUTINE GETNUM (NUM,UNIT)
        INTEGER NUM, UNIT
10      READ (UNIT,'(I10)', ERR=10) NUM
        RETURN
        END
```

3.2.41 The Type Statement

Syntax

type vname1 [, *vname2*]...

Purpose

Specifies the type of user-defined names.

Remarks

type is one of the following data type specifiers:

INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
COMPLEX
COMPLEX*8
COMPLEX*16
LOGICAL
LOGICAL*2
LOGICAL*4
CHARACTER
CHARACTER*n

vname is the symbolic name of a variable, array, or statement function; or a function subprogram or an array declarator. A type statement can confirm or override the implicit type of a name. A type statement can also specify dimension information.

n (as in CHARACTER*n) may be in the following range (1 <= *n* <= 127).

A user name for a variable, array, external function, or statement function may appear in a type statement. Such an appearance defines the type of that name for the entire program unit. Within a program unit, a name can have its type explicitly specified by a type statement only once.

A type statement may also confirm the type of an intrinsic function, but it is not required. The name of a subroutine or main program cannot appear in a type statement.

The following rules apply to a type statement:

1. A type statement must precede all executable statements.
2. The data type of a symbolic name can be declared explicitly only once.
3. A type statement cannot be labeled.
4. A type statement can be used to declare an array by appending a dimension declarator to an array name.

A symbolic name can be followed by a data type length specifier of the form **length*, where *length* is one of the acceptable lengths for the data type being declared. Such a specification overrides the length attribute that the statement implies and assigns a new length to the specified item. If both a data type length specifier and an array declarator are included, the data type length specifier goes last.

Example

```
C EXAMPLE OF TYPE STATEMENTS
      INTEGER COUNT, MATRIX(4,4), SUM
      REAL MAN,IABS
      LOGICAL SWITCH
        .
      INTEGER*2 Q, M12*4, IVEC(10)*4
      REAL*4 WX1, WX3*4, WX5, WX6*4
        .
      CHARACTER NAME*10, CITY*80, CH
```

3.2.42 The WRITE Statement

Syntax

WRITE (*unit-spec* [, *format-spec*]
[, IOSTAT=*iocheck*] [, ERR=*slabel*]
[, REC=*rec-num*]) *iolist*

Purpose

Transfers data from the *iolist* items to the file associated with the specified unit.

Remarks

unit-spec is a required unit specifier and must appear as the first argument. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

format-spec is a format specifier. It is required as the second argument for a formatted WRITE; it must not appear for an unformatted WRITE.

The remaining arguments, if present, may appear in any order.

iocheck is an integer variable or integer array element that becomes defined as (1) a zero if no error or end of file conditions are encountered or (2) a processor-dependent positive integer value if an error condition is encountered or (3) a processor-dependent negative integer value if an end of file is encountered and no error condition exists. (See Section 3.1.7, "I/O Statements," for more information on error-handling.)

slabel is an optional statement label. If it is not present, I/O errors result in runtime errors. If it is present, I/O errors transfer control to the executable statement specified.

rec-num is a record number, specified for direct access files only (otherwise, an error results). It is a positive integer expression, specifying the number of the record to be written. The first record in the file is record number 1. If the record number is omitted for a direct access file, writing continues from the current position in the file.

iolist specifies the entities whose values are transferred by the WRITE statement. An *iolist* may be empty, but ordinarily consists of output entities and implied DO lists, separated by commas.

If the WRITE is internal, the character variable or character array element specified as the unit is the destination of the output; otherwise, the external unit is the destination.

If the file has not been opened by an OPEN statement, an implicit open operation is performed. The OPEN operation is equivalent to the following statement:

```
OPEN (unit-spec, FILE=' ', STATUS='NEW',
+ACCESS='SEQUENTIAL', FORM=format)
```

format is FORMATTED for a formatted WRITE statement and UNFORMATTED for an unformatted WRITE statement. See Section 3.2.30, "The OPEN Statement," for a description of the effect of the FILE= argument.

Example

```
C Display message: "One= 1,Two= 2,Three= 3"
C on the screen, not doing
C things in the simplest way!
      WRITE(*,980)' One=',1,1+1,'ee=',+(1+1+1)
980   FORMAT(A,I2,',Two=',1X,I1,',Thr',A,I2)
```

Chapter 4
The I/O System

This chapter supplements the presentation of the I/O statements in Chapter 3, "Statements." It describes the elements of the MS-FORTRAN file system, defines the basic concepts of I/O records and I/O units, and discusses the various kinds of file access available. It further relates these definitions to how various tasks are accomplished using the most common forms of files and I/O statements. The chapter includes a complete program illustrating the I/O statements and discusses general I/O system limitations.

4.1 Records

The building block of the MS-FORTRAN file system is the record. A record is a sequence of characters or values. There are three kinds of records: formatted, unformatted, and endfile.

1. Formatted

   A formatted record is a sequence of characters terminated by a system-dependent end-of-line marker. Formatted records are interpreted in a manner consistent with the way most operating systems and editors interpret lines.

2. Unformatted

   An unformatted record is a sequence of values in a system dependent form. Unformatted files contain a structure that defines the physical record. Binary files contain only the values written to them, and the record structure cannot, in general, be determined from this information.

3. Endfile

   The MS-FORTRAN file system simulates a virtual endfile record after the last record in a file. The way end of file is represented depends in part on the operating system.

4.2 Files

A file is a sequence of records. Files are either external or internal.

1. External

   An external file is either a file on a device or the device itself.

2. Internal

   An internal file is a character variable or character array element that serves as the source or destination of some formatted I/O operation.

For the remainder of this manual, both internal MS-FORTRAN files and the files known to the operating system are usually referred to simply as "files," with context determining meaning. The OPEN statement provides the link between the two notions of files; in most cases, the ambiguity disappears after opening a file, when the two notions coincide.

4.2.1 File Properties

A FORTRAN file has the following properties:

1. name
2. position
3. structure (formatted, unformatted, or binary)
4. access method (sequential or direct)

4.2.1.1 Filename

A file can have a name. If present, a name is a character string identical to the name by which the file is known to the operating system. Filenaming conventions are determined by your operating system.

4.2.1.2 File Position

The position of a file is usually set by the previous I/O operation. A file has an initial point, terminal point, current record, preceding record, and next record.

It is possible to be between records in a file, in which case the next record is the successor to the previous record, and there is no current record.

Opening a sequential file positions the file at its beginning. If the next I/O operation is a WRITE, all old data in the file is discarded. The file position after sequential WRITEs is at the end of the file, but not beyond the endfile record.

Executing the ENDFILE statement positions the file beyond the endfile record, as does a READ statement executed at the end of the file. You can detect the endfile condition by using the END= option in a READ statement.

4.2.1.3 File Structure

An external file may be opened as a formatted, unformatted, or binary file. All internal files are formatted.

1. Formatted

   Files consisting entirely of formatted records.

2. Unformatted

   Files consisting entirely of unformatted records.

3. Binary

   Sequences of bytes with no internal structure.

4.2.1.4 File Access Method

An external file is opened as either a sequential file or a direct access file.

1. Sequential

   Files that contain records whose order is determined by the order in which the records were written (the normal sequential order). These files must not be read or written using the REC= option, which specifies a position for direct access I/O.

2. Direct

   Files whose records can be read or written in any order (they are random access files). Records are numbered sequentially, with the first record numbered 1. All records have the same length, specified when the file is opened; each record has a unique record number, specified when the record is written.

In a direct access file, it is possible to write records out of order (e.g., 9, 5, and 11 in that order), without writing the records in between. It is not possible to delete a record once written; however, a record can be overwritten with a new value.

Reading a record that has not been written from a direct access file will result in an error. Direct access files must reside on disk. The operating system attempts to extend direct access files if a record is written beyond the old terminating file boundary; the success of this operation depends on the existence of room on the physical device.

4.2.2 Special Properties of Internal Files

An internal file is a character variable or character array element. The file has exactly one record, which is of the same length as the character variable or character array element.

If less than the entire record is written, the remaining portion of the record is filled with blanks. The file position is always at the beginning of the file prior to execution of the I/O statement. Internal files permit only formatted, sequential I/O; and only the I/O statements READ and WRITE may specify an internal file.

Internal files provide a mechanism for using the formatting capabilities of the I/O system to convert values to and from their external character representations and their MS-FORTRAN internal memory representations. That is, reading from an internal file converts the character values into numeric, logical, or character values; writing to an internal file converts values into their (external) character representation. The backslash edit descriptor (\) may not be used with internal files.

4.2.3 Units

A unit is a means of referring to a file. A unit specified in an I/O statement is either an external unit specifier or an internal unit specifier.

1. External unit specifier

   An external unit specifier is either an integer expression or the character * (which stands for the screen, for writing, and the keyboard, for reading).

   In most cases, an external unit specifier value is bound to a physical device (or files resident on the device) by name, using the OPEN statement. Once this binding of a unit to a system filename occurs, MS-FORTRAN I/O statements specify the unit number to refer to the associated external entity. Once the file is opened, the external unit specifier value is uniquely associated with a particular external entity until an explicit CLOSE operation occurs or until the program terminates.

   The only exception to these binding rules is that the unit value zero is initially associated with the keyboard for reading and the screen for writing and no explicit OPEN statement is necessary. The MS-FORTRAN file system interprets the character * as unit zero.

2. Internal unit specifier

   An internal unit specifier is a character variable or character array element that directly specifies an internal file.

See Section 4.3.1, "Elements of I/O Statements," for a discussion of how these unit specifiers are used.

4.2.4 Commonly Used File Structures

Numerous combinations of file structures are possible in MS-FORTRAN. However, two kinds of files suffice for most applications:

1. * files
2. named, external, sequential, formatted files

* represents the keyboard and screen, that is, a sequential, formatted file, also known as unit zero. When reading from unit zero, you must enter an entire line; the normal operating system conventions for correcting typing mistakes apply.

An external file can be bound to a system name by any one of the following methods:

1. If the file is explicitly opened, the name can be specified in the OPEN statement.
2. If the file is explicitly opened and the name is specified as all blanks, the name is read from the command line (if available). If the command line is unavailable or contains no name, the user will usually be prompted for the name.
3. If the file is implicitly opened (with a READ or WRITE statement) the name is obtained as in method 2, described in the preceding paragraph.
4. If the file is explicitly opened and no name is specified in the OPEN statement, the file is considered a scratch or temporary file, and an implementation-dependent name is assumed.

The following sample program uses * files and named, external, sequential, formatted files for reading and writing. The I/O statements themselves are explained in general in Section 4.3, "I/O Statements." For details of each individual I/O statement, see the appropriate entries in Section 3.2, "Statement Directory."

```
C COPY A FILE WITH THREE COLUMNS OF INTEGERS,
C EACH 7 COLUMNS WIDE, FROM A FILE WHOSE NAME
C IS ENTERED BY THE USER TO ANOTHER FILE NAMED
C OUT.TXT, REVERSING THE POSITIONS OF THE
C FIRST AND SECOND COLUMNS.
            PROGRAM COLSWP
            CHARACTER*64 FNAME

C PROMPT TO THE SCREEN BY WRITING TO *.
            WRITE(*,900)
900         FORMAT(' INPUT FILE NAME -'\)

            C READ THE FILE NAME FROM THE KEYBOARD BY
            C READING FROM *.
            READ(*,910) FNAME
910         FORMAT(A)

C USE UNIT 3 FOR INPUT; ANY UNIT NUMBER EXCEPT
C 0 WILL DO.
            OPEN(3,FILE=FNAME)

C USE UNIT 4 FOR OUTPUT; ANY UNIT NUMBER EXCEPT
C 0 AND 3 WILL DO.
            OPEN(4,FILE='OUT.TXT',STATUS='NEW')

C READ AND WRITE UNTIL END OF FILE.
100         READ(3,920,END=200)I,J,K
            WRITE(4,920)J,I,K
920         FORMAT(3I7)
            GOTO 100
200         WRITE(*,910)'Done'
            END
```

4.2.5 Other File Structures

The less commonly used file structures are appropriate for certain classes of applications. A very general indication of their intended uses follows:

1. If random access I/O is needed, as would probably be the case in a data base, direct access files are necessary.
2. If the data is to be both written and reread by MS-FORTRAN, unformatted files are perhaps more efficient in terms of speed, but possibly less efficient in terms of disk space. The combination of direct and unformatted files is ideal for a data base created, maintained, and accessed exclusively by MS-FORTRAN.

3. If the data must be transferred without any system interpretation, especially if all 256 possible byte values are to be transferred, unformatted I/O is necessary.

   One use of unformatted I/O is in the control of a device that has a single-byte, binary interface. Formatted I/O would, in this example, interpret certain characters, such as the ASCII representation for RETURN, and fail to pass them through to the program unaltered.

   The number of bytes written for an integer constant is determined by the $STORAGE metacommand (for details, see Section 6.2.12, "The $STORAGE Metacommand.")

4. If the data is to be transferred as in the third use described in this list, but will be read by non-FORTRAN programs, the BINARY format is recommended. Unformatted files are blocked internally, and consequently the non-FORTRAN program must be compatible with this format to interpret the data correctly. BINARY files contain only the data written to them. Backspacing over records is not possible and incomplete records cannot be read from them.

4.2.6 OLD and NEW Files

A file opened in MS-FORTRAN is either OLD or NEW, but "opened for reading" is not distinguishable from "opened for writing." Therefore, you can open OLD (existing) files and write to them, with the effect of overwriting them.

Similarly, you can alternately WRITE and READ to the same file (providing that you avoid reading beyond the end of the file, or reading unwritten records in a direct file). A WRITE to a sequential file effectively deletes any records that existed beyond the newly written record.

When a device such as the keyboard or printer is opened as a file, it normally makes no difference whether it is opened as OLD or NEW. With disk files, however, opening a file as NEW creates a new file:

1. If a previous file existed with the same name, the previous file is deleted.

2. If the new file is closed with STATUS='KEEP' or if the program terminates without doing a CLOSE operation on that file, a permanent file is created with the name given when the file was opened.

4.2.7 Limitations

Certain limitations on the use of the MS-FORTRAN I/O system are described briefly in the following list:

1. Direct files/direct device association

   There are two kinds of devices: sequential and direct. The files associated with sequential devices are streams of characters; except for reading and writing, no explicit motion is allowed. The keyboard, screen, and printer are all sequential devices.

   Direct devices, such as disks, have the additional task of seeking a specific location. Direct devices can be accessed either sequentially or randomly, and thus can support direct files. The MS-FORTRAN I/O system does not allow direct files on sequential devices.

2. BACKSPACE/BINARY sequential file association

   There is no indication in a binary sequential file of record boundaries; therefore, a BACKSPACE operation on such files is defined as backing up by one byte. Direct files contain records of fixed, specified length, so it is possible to backspace by records on direct unformatted files.

3. Partial READ/BINARY file

   The data read from a binary file must correspond in length to the data written. Unformatted sequential files differ, in that an internal structure allows part or none of a record to be read (the unread part is skipped).

4. Side effects of functions called in I/O statements

   During execution of any I/O statement, evaluation of an expression may cause a function to be called. That function call must not cause any I/O statement to be executed.

4.3 I/O Statements

This section discusses the elements of I/O statements in general. For specific details on each of the seven I/O statements OPEN, CLOSE, READ, WRITE, BACKSPACE, ENDFILE, and REWIND, see the appropriate entries in Section 3.2, "Statement Directory," in the previous chapter.

In addition to these I/O statements, there is an I/O intrinsic function, EOF(*unit-spec*), which is described in Section 5.3.2, "Intrinsic Functions." EOF returns a logical value that indicates whether there is any data remaining in the file after the current position.

4.3.1 Elements of I/O Statements

The various I/O statements take certain arguments that specify sources and destinations of data transfer as well as other facets of the I/O operation. The elements described in this subsection are the following:

1. unit specifier (*unit-spec*)
2. format specifier (*format-spec*)
3. input/output list (*iolist*)

4.3.1.1 The Unit Specifier

The unit specifier, *unit-spec*, can take one of the following forms in an I/O statement:

1. The * specifier

   ```
   WRITE (*,*) 'Begin output.'
   ```

   The first (*) in this example refers to the keyboard or screen and specifies that unit.

2. Integer expression

```
WRITE (10,*) 'File 10:'
```

   The integer (10) refers to an external file associated with unit 10. An (*) indicates a unit number zero. Unit specifier numbers in the range -32767 to 32767 are accepted.

3. Name of a character variable or character array element

```
CHARACTER*10 STRING
WRITE (STRING, '(I10)') IVAL
```

   The character variable, STRING, refers to an internal file.

See Section 4.2.3, "Units," for a discussion of the difference between external and internal unit specifiers.

4.3.1.2 Format Specifiers In I/O Statements

The format specifier, *format-spec*, can take one of the following forms in an I/O statement:

1. FORMAT Statement label

```
          WRITE (*,990) I,J,K
990       FORMAT (1X,215,13)
```

   The statement label 990 refers to the FORMAT statement at 990.

2. Integer variable name

```
          ASSIGN 990 TO IFMT
990       FORMAT(1X,2I5,I3)
          WRITE(*,IFMT) I,J,K
```

   In the WRITE statement, the integer variable name (IFMT) refers to FORMAT statement label 990 as assigned just before the FORMAT statement. For further information, see Section 3.2.1, "The ASSIGN Statement."

3. Character expression

```
WRITE(*,'(1X,2I5,I3)')I,J,K
```

   The value of the character expression is the format for the data transfer.

4. Character variable

```
CHARACTER*11 FMTCH
FMTCH = '(1X,2I5,I3)'
WRITE(*,FMTCH)I,J,K
```

In this example, the WRITE statement uses the contents of the character variable FMTCH as the format specifier for data transfer.

5. *

```
WRITE(*,*) I,J,K
```

In this statement, the second asterisk indicates a list-directed I/O transfer. For more information, see Section 4.5, "List-Directed I/O."

4.3.1.3 Input/Output List

The input/output list, *iolist*, specifies the entities whose values are transferred by READ and WRITE statements. An *iolist* may be empty, but ordinarily consists of input or output entities and implied DO lists, separated by commas.

An input entity can be specified in the *iolist* of a READ statement and an output entity in the *iolist* of a WRITE statement.

1. Input entities

   An input entity is either a variable name, an array element name, or an array name. An array name specifies all of the elements of the array in memory sequence order.

2. Output entities

   In addition to being any of the items listed as input entities, an output entity can be any other expression not beginning with the left parenthesis character "(". (The left parenthesis distinguishes implied DO lists from expressions.)

To distinguish it from an implied DO list, the following expression

```
(A+B)*(C+D)
```

can be written as:

```
+(A+B)*(C+D)
```

3. Implied DO lists

   Implied DO lists can be specified as items in the I/O list of READ and WRITE statements and have the following format:

   (*iolist*, *variable* = *expr1*,
   expr2 [, *expr3*])

   iolist is defined the same as for elements of I/O statements (including nested implied DO lists).

   variable, *expr1*, *expr2*, and *expr3* are the same as defined for the DO statement. That is, *variable* is an integer variable, while *expr1*, *expr2*, and *expr3* are integer expressions.

   In a READ statement, the DO variable (or an associated entity) must not appear as an input list item in the embedded *iolist*, but may have been read in the same READ statement before the implied DO list. The embedded *iolist* is effectively repeated for each iteration of *variable* with appropriate substitution of values for the DO variable.

   In the case of nested implied DO loops, the innermost (most deeply nested) loop is always executed first.

4.3.2 Carriage Control

The first characters of records written to files are treated as other characters in the record. However, first characters in records transferred to unit 0 or *, or to the files PRN, LPT1, or CON, are not printed. These characters are not interpreted as carriage control characters. The MS-FORTRAN I/O system recognizes certain characters as carriage control characters. These characters and their effects when printed are shown in Table 4.1.

Table 4.1
Carriage Control Characters

Character	Effect
space	Advances one line.
0	Advances two lines.
1	Advances to top of next page (ignored by the console).
+ (plus)	Does not advance (allows overprinting).

Any character other than those listed in the preceding table is treated as a space and deleted from the print line. If you accidentally omit the carriage control character, the first character of the record is not printed.

4.4 Formatted I/O

If a READ or WRITE statement specifies a format, the I/O statement is considered a formatted, rather than an unformatted, I/O statement (see Section 4.3.1, "Elements in I/O Statements," for more information on format specification.)

The following five examples are all valid and equivalent means of specifying a format in an I/O statement and are a review of the format specifiers listed in Section 4.3.1.

```
(1)          WRITE (*,990) I,J,K
      990    FORMAT(1X,2I5,I3)

(2)          ASSIGN 990 TO IFMT
      990    FORMAT(1X,2I5,I3)
             WRITE(*,IFMT) I,J,K

(3)          WRITE(*,'(1X,2I5,I3)')I,J,K

(4)          CHARACTER*11 FMTCH
             FMTCH = '(1X,2I5,I3)'
             WRITE(*,FMTCH)I,J,K

(5)          WRITE(*,*) I,J,K
```

The format specification must begin with a left parenthesis character and end with a matching right parenthesis character. The leading left parenthesis can be preceded by initial blank characters. Characters beyond the matching right parenthesis are ignored.

See Section 4.4.2, "Edit Descriptors", and Section 4.5, "List-Directed I/O", for more details on format editing for data transfer.

4.4.1 Interaction Between Format and I/O List

If an *iolist* contains at least one item, at least one repeatable edit descriptor must exist in the format specification. In particular, the empty edit specification, ( ), can be used only if no items are specified in the *iolist* (in which case a WRITE writes a zero length record and a READ skips to the next record).

Each item in the *iolist* is associated with a repeatable edit descriptor during the I/O statement execution. In contrast, the remaining format control items interact directly with the record and do not become associated with items in the *iolist*.

Note

Two repeatable edit descriptors are required in the FORMAT statement or format descriptor for each COMPLEX data item in the *iolist*.

The items in a format specification are interpreted from left to right. Repeatable edit descriptors act as if they were present *r* times (if omitted, *r* is treated as a repeat factor of one). A format specification itself can have a repeat factor, as in:

```
10(5F10.4, 2(3x,5I3))
```

During the formatted I/O process, the "format controller" scans and processes the format items as described in the previous paragraph. When a repeatable edit descriptor is encountered, one of the following occurs:

1. A corresponding item appears in the *iolist*, in which case the item and the edit descriptor are associated and I/O of that item proceeds under format control of the edit descriptor.
2. No corresponding item appears in the *iolist*, in which case the format controller terminates I/O. Thus, for the following statements:

```
      I=5
      WRITE (*,10) I
10    FORMAT (1X,'I= ',I5,'J= ',I5)
```

the output would look like this:

```
I=          5J=
```

If the format controller encounters the matching final right parenthesis of the format specification, or a colon (:) edit descriptor, and if there are no further items in the *iolist*, the format controller terminates I/O.

If, however, there are further items in the *iolist*, the file is positioned at the beginning of the next record and the format controller continues by rescanning the format, starting at the beginning of the format specification terminated by the last preceding right parenthesis.

If there is no such preceding right parenthesis, the format controller rescans the format from the beginning. Within the portion of the format rescanned, there must be at least one repeatable edit descriptor.

If the rescan of the format specification begins with a repeated nested format specification, the repeat factor indicates the number of times to repeat that nested format specification. The rescan does not change the previously set scale factor or the BN or BZ blank control in effect.

When the format controller terminates, the remaining characters of an input record are skipped or an end-of-record is written on output. An exception to this occurs when the backslash edit descriptor (\) is used.

4.4.2 Edit Descriptors for the FORMAT Statement

Edit or format descriptors in FORTRAN specify the form of a record and control the editing between the characters in a record and the internal format of data. There are two types of edit descriptors: repeatable and nonrepeatable. Both are described in the following sections of this chapter.

4.4.2.1 Nonrepeatable Edit Descriptors

1. Apostrophe (') editing ('xxxx')

   The apostrophe edit descriptor has the form of a character constant and causes the character constant to be transmitted to the output unit. Embedded blanks are significant; two adjacent apostrophes, i.e., single right quotation marks, must be used to represent a single apostrophe within a character constant. Apostrophe editing cannot be used for input (READ). For an example, see "Hollerith editing (H)."

2. Hollerith editing (H)

   The *n*H edit descriptor transmits the next *n* characters, with blanks counted as significant, to the output unit. Hollerith editing cannot be used for input (READ).

   Examples of apostrophe and Hollerith editing:

```
C EACH WRITE OUTPUTS CHARACTERS
C BETWEEN THE SLASHES: /ABC'DEF/

C APOSTROPHE EDITING
      WRITE (*,970)
970   FORMAT (' ABC''DEF')
      WRITE (*,'('' ABC''''DEF'')')

C SAME OUTPUT USING HOLLERITH EDITING
      WRITE (*,'(8H ABC''DEF)')
      WRITE (*,960)
960   FORMAT (8H ABC'DEF)
```

   The leading blank in each case in the preceding examples is a carriage control character to cause a line feed (carriage return) on output.

3. Positional editing (T*c*, TL*c*, TR*c*)

   The T, TL,and TR edit descriptors specify the position in the record to which or from which the next character will be transmitted. The position specified by a T edit descriptor may be in either direction from the current position. This allows a record to be processed more than once on input. On output, the character positions not specified by the T, TL, and TR edit descriptors are filled with blanks as if the record were initially filled with blanks.

   The T*c* edit descriptor specifies that the transmission of the next character is to occur at the *c*th character position. The TR*c* edit descriptor specifies that the transmission of the next character is to occur at *c* characters forward from the current position. The TL*c* edit descriptor specifies that the transmission of the next character is to occur at *c* characters backward from the current position.

Note

If the current position is less than or equal to the value of *c*, TL*c* editing will cause transmission to or from position one of the current record.

You may not use the T descriptors to reposition to the left once you have positioned beyond position 128 since the output data are held in a buffer of this size.

---

4. Positional editing (X)

   On input (READ), the *n*X edit descriptor advances the file position *n* characters, skipping *n* characters. On output (WRITE), the *n*X edit descriptor writes *n* blanks.

5. Optional plus editing (SP, SS and S)

   The SP, SS, and S edit descriptors can be used to control optional plus characters in numeric output fields. SP causes output of the plus sign in all subsequent positions that the processor recognizes as optional plus fields. SS causes plus sign suppression in all subsequent positions that the processor recognizes as optional plus fields. S restores the default for producing the optional plus sign.

6. Slash editing (/)

   The slash indicates the end of data transfer on the current record. On input, the file is positioned to the beginning of the next record. On output, an end-of-record is written, and the file is positioned to write on the beginning of the next record.

7. Backslash editing (\)

   Normally when the format controller terminates, the end of data transmission on the current record occurs. If the last edit descriptor encountered by the format controller is a backslash (\), this automatic end-of-record is inhibited, allowing subsequent I/O statements to continue reading (or writing) from (or to) the same record.

   This mechanism is most commonly used to prompt to the screen and read a response from the same line, as in the following example:

```
WRITE (*,'(A\)') 'Input an integer - ->'
READ (*,'(BN,I6)') I
```

   The backslash edit descriptor does not inhibit the automatic end-of-record generated when reading from the * unit; input from the keyboard must always be terminated by the RETURN key. The backslash edit descriptor may not be used with internal files.

8. Terminating format control (:)

   The colon (:) edit descriptor terminates format control if there are no more items in the *iolist*. This tool can be used to suppress output when some of the characters in the format do not have corresponding data in the *iolist*.

9. Scale factor editing (P)

   The *k*P edit descriptor sets the scale factor for subsequent F and E edit descriptors until the next *k*P edit descriptor. At the start of each I/O statement, the scale factor is initialized to zero. The scale factor affects format editing in the following ways:

   a. On input, with F and E editing (providing that no explicit exponent exists in the field) and F output editing, the externally represented number equals the internally represented number multiplied by 10**k.

b. On input, with F and E editing, the scale factor has no effect if there is an explicit exponent in the input field.

c. On output, with E editing, the real part of the quantity is output multiplied by 10**k and the exponent is reduced by k (effectively altering the column position of the decimal point but not the value output).

10. Blank interpretation (BN and BZ)

These edit descriptors specify the interpretation of blanks in numeric input fields. The default, BZ, is set at the start of each I/O statement. This makes blanks, other than leading blanks, identical to zeros. If MS-FORTRAN processes a BN edit descriptor, however, blanks in subsequent input fields are ignored unless, and until, a BZ edit descriptor is processed.

The effect of ignoring blanks is to take all the nonblank characters in the input field and treat them as if they were right-justified in the field with the number of leading blanks equal to the number of ignored blanks. For instance, the following READ statement accepts the characters shown between the slashes as the value 123. RETURN indicates a carriage return or enter keystroke.

```
      READ(*,100) I
100   FORMAT (BN,I6)

      /123          RETURN/
      /123       456RETURN/
      /  123        RETURN/
```

BN editing will go into effect automatically when a READ is associated with a "short" record; "short" meaning that the total number of characters in the input record is fewer than the combined number of characters specified by the format descriptors and *iolist*. The record is padded on the right with blanks to the required length. Thus, the following example would result in the value 123, rather than 12300. RETURN represents a carriage return or entry keystroke.

```
READ (*,'(I5)') I
/123RETURN/
```

The BN edit descriptor, in conjunction with the infinite blank padding at the end of formatted records, makes interactive input very convenient.

4.4.2.2 Repeatable Edit Descriptors

The I, F, E, D, and G edit descriptors are used for I/O of numeric data. The following general rules apply to all numeric edit descriptors:

1. On input, leading blanks are not significant. Other blanks are interpreted differently depending on the BN or BZ flag in effect, but fields that are all blank always become the value zero. Plus signs are optional. The blanks supplied by the file system to pad a record to the required size are also not significant.
2. On input with F, E, D and G editing, an explicit decimal point appearing in the input field overrides the edit descriptor specification of the decimal point position.
3. On output, the characters generated are right-justified in the field and padded by leading blanks, if necessary.
4. On output, if the number of characters produced exceeds the field width or the exponent exceeds its specified width, the entire field is filled with asterisks.
5. When reading with I, F, E, D, G or L edit descriptors, the input field may contain a comma, which is considered to terminate the field. Reading of the next field will start at the character following the comma. The missing characters are not significant. For example,

```
READ (*,'(3I5)') I,J,K
/1,  2  ,     3,.....
```

will result in I=1, J=20 and K=3.

Note

Do not use this feature if you wish to rely on explicit field position editing (i.e., using the T, TL or TR edit descriptors.)

Individual descriptions of the repeatable edit descriptors follow.

1. Integer editing (I)

   The edit descriptor I*w* must be associated with an *iolist* item of type INTEGER. The field is *w* characters wide. On input, an optional sign may appear in the field.

2. F real editing

   The edit descriptor F*w.d* must be associated with an *iolist* item of type REAL or REAL*8. The field is *w* characters wide, with a fractional part *d* digits wide. The input field begins with an optional sign followed by a string of digits which may contain an optional decimal point. If the decimal point is present, it overrides the *d* specified in the edit descriptor; otherwise, the rightmost *d* digits of the string are interpreted as following the decimal point (with leading blanks converted to zeros, if necessary). Following this is an optional exponent which is either:

   a. + (plus) or - (minus) followed by an integer, or

   b. E followed by zero or more blanks followed by an optional sign followed by an integer.

   The output field occupies *w* digits, *d* of which fall beyond the decimal point. The value output is controlled both by the *iolist* item and the current scale factor. The output value is rounded rather than truncated.

3. E and D real editing

   The E edit descriptor takes one of the forms E*w.d* or E*w.d*E*e*. The D edit descriptor takes the form D*w.d*. All parameters and rules for the E edit descriptor apply to the D edit descriptor.

   For each form, the field is *w* characters wide. The *e* has no effect on input. The input field for the E and D edit descriptors is identical to that described by an F edit descriptor with the same *w* and *d*.

   The form of the output field depends on the scale factor (set by the P edit descriptor) in effect. For a scale factor of zero, the output field is a minus sign (if necessary), followed by a decimal point, followed by a string of digits, followed by an exponent field for exponent *exp*, of one of the forms shown in the list of scale factors that follows.

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
E*w.d*	|*exp*| <= 99	E followed by plus or minus, followed by the two-digit exponent
E*w.d*	99 < |*exp*| <= 999	Plus or minus, followed by the three-digit exponent
E*w.d*E*e*	|*exp*| <= (10**e*)-1	E followed by plus or minus, followed by *e* digits which are the exponent with possible leading zeros
D*w.d*	|*exp*| <= 99	D followed by plus or minus, followed by the two-digit exponent
D*w.d*	99 < |*exp*| <= 999	Plus or minus, followed by the three-digit exponent

The forms E*w.d* and D*w.d* must not be used if the absolute value of the exponent to be printed exceeds 999.

The scale factor controls the decimal normalization of the printed E or D field. If the scale factor, *k*, is in the range (*-d*<*k*<=0), then the output field contains exactly *k* leading zeros after the decimal point and *d* + *k* significant digits after this. If (0<*k*<*d*+2), then the output field contains exactly *k* significant digits to the left of the decimal point and (*d*-*k*-1) places after the decimal point. Other values of *k* are errors.

4. G real editing

   The G edit descriptor takes the forms G*w.d* and G*w.d*E*e*. For either form, the input field is *w* characters wide, with a fractional part consisting of *d* digits. If the scale factor is greater than one, the exponent part consists of *e* digits.

   G input editing is the same as F input editing.

   G output editing is dependent on the magnitude of the data being edited. The following list illustrates the output equivalent for the magnitude of data.

Data Magnitude	Conversion Equivalent
M < 0.1	E*w.d*
0.1 <= M < 1	F(*w*-n).*d*,n('*b*'); where n is 4 for G*w.d*; n is *e*+2 for G*w.d*E*e*, and '*b*' represents a blank character.
1 <= M < 10	F(*w*-n).(*d*-1), n('*b*')
.	.
.	.
10**(*d*-2) <= M <10**(*d*-1)	F(*w*-n).1, n('*b*')
10**(*d*-1) <= M <10***d*	F(*w*-n).0, n('*b*')
M >= 10***d*	E*w.d*

5. Two successively interpreted edit descriptors of the types D, E, F, and G are used to specify the editing of complex numbers. The types may be used in combination. The first edit descriptor will specify the real part of the complex number; the second will specify the imaginary part.

Note

Nonrepeatable edit descriptors may appear between the D, E, F, and G descriptors.

---

6. Logical editing (L)

   The edit descriptor takes the form L*w*, indicating that the field is *w* characters wide. The *iolist* element associated with an L edit descriptor must be of type LOGICAL. On input, the field consists of optional blanks, followed by an optional decimal point, followed by T (for true) or F (for false). Any further characters in the field are ignored, but accepted on input, so that .TRUE. and .FALSE. are valid inputs. On output, w-1 blanks are followed by either T or F, as appropriate.

7. Character editing (A)

   The forms of the edit descriptor are A or A*w*. In the first form, A acquires an implied field width, *w*, from the number of characters in the *iolist* associated item. The *iolist* item may be of any type. If it is not of type CHARACTER, it is assumed to have one character per byte, so that its length is as specified in the list of data conversion equivalents just presented.

   On input, if *w* exceeds or equals the number of characters in the *iolist* element, the rightmost characters of the input field are used as the input characters; otherwise, the input characters are left-justified in the input *iolist* item and trailing blanks are provided.

   If the number of characters input is not equal to *w*, then the input field will be blankfilled or truncated on the right to the length of *w* before being transmitted to the *iolist* item. For example, if the following program fragment is executed,

```
CHARACTER*10 C
READ(*,'(A15)') C
```

   and the following thirteen characters are typed in at the keyboard,

```
'ABCDEFGHIJKLM'
```

   the input field will be filled to fifteen characters:

```
'ABCDEFGHIJKLM  '
```

Then the rightmost ten characters will be transmitted to the *iolist* element C:

```
'FGHIJKLM '
```

On output, if *w* exceeds the characters produced by the *iolist* item, leading blanks are provided; otherwise, the leftmost *w* characters of the *iolist* item are output.

4.5 List-Directed I/O

A list-directed record is a sequence of values and value separators.

Each value in a list-directed record is one of the following:

1. a constant
2. a null value
3. either a constant or a null value multiplied by an unsigned, nonzero, integer constant; that is, r*c (r successive appearances of the constant c) or r* (r successive null values). Except in string constants, none of these may have embedded blanks.

Each value separator in a list-directed record is one of the following:

1. a comma, optionally preceded or followed by one or more contiguous blanks
2. a slash, optionally preceded or followed by one or more contiguous blanks
3. one or more contiguous blanks between two constants, or after the last constant

4.5.1 List-Directed Input

Except as noted in the following list, input forms acceptable to format specifications for a given type are also acceptable for list-directed formatting.

The form of the input value must be acceptable for the type of the input list item. Never use blanks as zeros. Only use embedded blanks within character constants, as specified in the following list. Note that the end-of-record has the effect of a blank, except when it appears within a character constant.

1. Real or double precision constants

   A real or double precision constant must be a numeric input field; that is, a field suitable for F editing. It is assumed to have no fractional digits unless there is a decimal point within the field.

2. Complex constants

   The form of a complex constant is an ordered pair of real or integer constants separated by a comma and surrounded by an opening and a closing parenthesis. The first constant of the pair is the real part of the the complex constant and the second is the imaginary part.

3. Logical constants

   A logical constant must not include either slashes or commas among the optional characters permitted for L editing.

4. Character constants

   A character constant is a nonempty string of characters, enclosed in single quotation marks. Each single quotation mark within a character constant must be represented by two single quotation marks, with no intervening blank or end-of-record.

   Character constants may be continued from the end of one record to the beginning of the next; the end of the record doesn't cause a blank or other character to become part of the constant. The constant may be continued on as many records as needed and may include the characters blank, comma, and slash.

   If the length n of the list item is less than or equal to the length m of the character constant, the leftmost n characters of the latter are transmitted to the list item.

If *n* is greater than *m*, the constant is transmitted to the leftmost *m* characters of the list item. The remaining *n* minus *m* characters of the list item are filled with blanks.

The effect is the same as if the constant were assigned to the list item in a character assignment statement.

5. Null values

   You can specify a null value in one of three ways:

   a. no characters between successive value separators

   b. no characters preceding the first value separator in the first record read by each execution of a list-directed input statement

   c. the r* form (described at the beginning of Section 4.5, "List-Directed I/O")

   A null value has no effect on the definition status of the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains so.

   A slash encountered as a value separator during execution of a list-directed input statement stops execution of that statement after the assignment of the previous value. Any further items in the input list are treated as if they were null values.

6. Blanks

   All blanks in a list-directed input record are considered to be part of some value separator, except for the following:

   a. blanks embedded in a character constant

   b. leading blanks in the first record read by each execution of a list-directed input statement (unless immediately followed by a slash or comma)

4.5.2 List-Directed Output

The form of the values produced is the same as required for input, except as noted in the following list. The list-directed line size is 80 columns.

1. New records are created as necessary, but, except for character constants, neither the end of a record nor blanks will occur within a constant.
2. Logical output constants are T for the value true and F for the value false.
3. Integer output constants are produced with the effect of an I12 edit descriptor.
4. Real and double precision constants are produced with the effect of either an F or an E edit descriptor, depending on the value of x in the following range:

   ```
   10**0 <= x <= 10**7
   ```

   a. If x is within the range, the constant is produced using 0PF16.7 for single precision and 0PF23.14 for double precision.

   b. If x is outside the range, the constant is produced using 1PE14.6 for single precision and 1PE21.13 for double precision.

5. Character constants produced have the following characteristics:

   a. They are not delimited by apostrophes (single quotation marks).

   b. They are neither preceded nor followed by a value separator.

   c. Each internal apostrophe (single quotation mark) is represented by one externally.

   d. A blank character is inserted at the start of any record that begins with the continuation of a character constant from the preceding record.

6. Slashes, as value separators, and null values are not produced by list-directed formatting.
7. In order to provide carriage control when the record is printed, each output record begins with a blank character.

Chapter 5
Programs, Subroutines, and Functions

As described in Section 1.2, "Programs and Compilable Parts of Programs," a program unit is either a main program, a subroutine, block data subprogram, or a function. Functions and subroutines are collectively called subprograms, or procedures. The PROGRAM, SUBROUTINE, BLOCK DATA and FUNCTION statements, as well as the statement function statement, are described in detail in Section 3.2, "Statement Directory." Related information is provided in the entries for the CALL and RETURN statements.

This chapter supplements the discussion of these individual statements with information on types of functions and a description of the relationship between formal and actual arguments in a function or subroutine call.

5.1 Main Program

A main program is any program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. The first statement of a main program may be a PROGRAM statement. If the main program does not have a PROGRAM statement, it will be assigned the name MAIN. The name MAIN then cannot be used to name any other global entity.

The execution of a program always begins with the first executable statement in the main program. Consequently, there must be precisely one main program in every executable program.

For further information about programs, see Section 3.2.33, "The PROGRAM Statement."

5.2 Subroutines

A subroutine is a program unit that can be called from other program units with a CALL statement. When invoked, a subroutine performs the set of actions defined by its executable statements and then returns control to the statement immediately following the one that called it or to a statement specified as an alternate return (see Section 3.2.5, "The CALL Statement.")

A subroutine does not directly return a value, although values can be passed back to the calling program unit via arguments or common variables.

For further information about subroutines, see Section 3.2.40, "The SUBROUTINE Statement."

5.3 Functions

A function is referred to in an expression and returns a value that is used in the computation of that expression. There are three kinds of functions:

1. external functions
2. intrinsic functions
3. statement functions

Each of these is described in more detail in the following sections.

Reference to a function may appear in an arithmetic or logical expression. When the function reference is executed, the function is evaluated and the resulting value used as an operand in the expression that contains the function reference. The format of a function reference is as follows:

fname ([*arg* [, *arg*] ...])

fname is the user-defined name of an external, intrinsic, or statement function.

arg is an actual argument.

The rules for arguments for functions are identical to those for subroutines (except that alternate returns are not allowed) and are described in Section 3.2.4, "The CALL Statement." Some additional restrictions that apply for intrinsic functions and for statement functions are described in Section 5.3.2, "Intrinsic Functions," and Section 5.3.3, "Statement Functions," respectively.

5.3.1 External Functions

An external function is specified by a function program unit. It begins with a FUNCTION statement and ends with an END statement. It may contain any kind of statement other than a PROGRAM statement, FUNCTION statement, or a SUBROUTINE statement.

5.3.2 Intrinsic Functions

Intrinsic functions are predefined by the MS-FORTRAN language and available for use in an MS-FORTRAN program. Table 5.1 gives the name, definition, argument type, and function type for all of the intrinsic functions available in MS-FORTRAN, with additional notes following the table.

An IMPLICIT statement cannot alter the type of an intrinsic function. For those intrinsic functions that allow several types of arguments, all arguments in a single reference must be of the same type.

An intrinsic function name can appear in an INTRINSIC statement. An intrinsic function name also can appear in a type statement, but only if the type is the same as the standard type of that intrinsic function.

Arguments to certain intrinsic functions are limited by the definition of the function being computed. For example, the logarithm of zero is mathematically undefined, and therefore not permitted.

All angles in Table 5.1 are expressed in radians. All arguments in an intrinsic function reference must be of the same type. X and Y are REAL; Z is a COMPLEX; I and J are INTEGER; and C, C1, and C2 are character values. Numbers in square brackets in column 1 refer to the notes following the table.

Furthermore, REAL is equivalent to REAL*4, DOUBLE PRECISION is equivalent to REAL*8. If the specified type of the argument is INTEGER, the type may be INTEGER*2 or INTEGER*4. If the specified type of the function is INTEGER, the type will be the default integer determined by the $STORAGE metacommand. (For further information, see Section 6.2.12, "The $STORAGE Metacommand.")

Table 5.1
Intrinsic Functions

Name	Definition	Type of Argument	Type of Function
Type Conversion			
INT (generic)	Convert to integer	(all)	INTEGER
INT(X) [1]	Convert to integer	REAL*4	INTEGER
IFIX(X)	Convert to integer	REAL*4	INTEGER
IDINT [2]	Convert to integer	REAL*8	INTEGER
REAL (generic)	Convert to REAL*4	(all)	REAL*4
REAL(I) [2]	Convert to REAL*4	INTEGER	REAL*4
DREAL(Z)	Return REAL*8 of COMPLEX*16	COMPLEX*16	REAL*8
FLOAT(I)	Convert to REAL*4	INTEGER	REAL*4
SNGL(X)	Convert to REAL*4	REAL*8	REAL*4
DBLE (generic)[3]			
	Convert to REAL*8	(all)	REAL*8
CMPLX(Z,[Y])[4]	Convert to COMPLEX*8	(all)	COMPLEX*8
DCMPLX(Z,[Y])	Convert to COMPLEX*16	(all)	COMPLEX*16
ICHAR(C) [5]	Convert to integer	CHARACTER	INTEGER
CHAR(X) [5]	Convert to character	INTEGER	CHARACTER
Truncation			
AINT (generic)	Truncate to REAL	REAL*4	REAL*4
		REAL*8	REAL*8
AINT(X)	Truncate to REAL*4	REAL*4	REAL*4
DINT(X)	Truncate to REAL*8	REAL*8	REAL*8
Nearest Whole Number			
ANINT (generic)	Round to REAL	REAL*4	REAL*4
		REAL*8	REAL*8
ANINT(X)	Round to REAL*4	REAL*4	REAL*4
DNINT(X)	Round to REAL*8	REAL*8	REAL*8
Nearest Integer			
NINT (generic)	Round to integer	REAL*4	INTEGER
		REAL*8	INTEGER
NINT(X)	Round to integer	REAL*4	INTEGER
IDNINT(X)	Round to integer	REAL*8	INTEGER

Table 5.1 *(continued)*

Name	Definition	Type of Argument	Type of Function
Absolute Value			
ABS (generic)	Absolute value	(all)	(all)
IABS(I)	INTEGER absolute	INTEGER	INTEGER
ABS(X)	REAL*4 absolute	REAL*4	REAL*4
DABS(X)	REAL*8 absolute	REAL*8	REAL*8
CABS(Z)	COMPLEX absolute	COMPLEX*8	REAL*4
		COMPLEX*16	REAL*8
CDABS(Z)	COMPLEX*16 absolute	COMPLEX*16	REAL*8
Remaindering			
MOD (generic)	Remainder	INTEGER	INTEGER
		REAL*4	REAL*4
		REAL*8	REAL*8
MOD(I,J)	INTEGER remainder	INTEGER	INTEGER
AMOD(X,Y)	REAL*4 remainder	REAL*4	REAL*4
DMOD(X,Y)	REAL*8 remainder	REAL*8	REAL*8
Transfer of Sign			
SIGN (generic)	Transfer of sign	INTEGER	INTEGER
		REAL*4	REAL*4
		REAL*8	REAL*8
ISIGN(I,J)	INTEGER transfer	INTEGER	INTEGER
SIGN(X,Y)	REAL*4 transfer	REAL*4	REAL*4
DSIGN(X,Y)	REAL*8 transfer	REAL*8	REAL*8
Positive Difference [5]			
DIM (generic)	Positive difference	INTEGER	INTEGER
		REAL*4	REAL*4
		REAL*8	REAL*8
IDIM(I,J)	INTEGER difference	INTEGER	INTEGER
DIM(X,Y)	REAL*4 difference	REAL*4	REAL*4
DDIM(X,Y)	REAL*8 difference	REAL*8	REAL*8
Choosing Largest Value			
MAX (generic)	Return maximum	INTEGER	INTEGER
		REAL*4	REAL*4
		REAL*8	REAL*8
MAX0(I,J,...)	INTEGER maximum	INTEGER	INTEGER
AMAX1(X,Y,...)	REAL*4 maximum	REAL*4	REAL*4
AMAX0(I,J,...)	REAL*4 maximum	INTEGER	REAL*4
MAX1(X,Y,...)	INTEGER maximum	REAL*4	INTEGER
DMAX1(X,Y,...)	REAL*8 maximum	REAL*8	REAL*8

Table 5.1 *(continued)*

Name	Definition	Type of Argument	Type of Function
Choosing Smallest Value			
MIN (generic)	Return minimum	INTEGER REAL*4 REAL*8	INTEGER REAL*4 REAL*8
MIN0(I,J,...)	INTEGER minimum	INTEGER	INTEGER
AMIN1(X,Y,...)	REAL*4 minimum	REAL*4	REAL*4
AMIN0(I,J,...)	REAL*4 minimum	INTEGER	REAL*4
MIN1(X,Y,...)	INTEGER minimum	REAL*4	INTEGER
DMIN1(X,Y,...)	REAL*8 minimum	REAL*8	REAL*8
REAL*8 Product			
DPROD(X,Y)	REAL*8 product	REAL*4	REAL*8
Imaginary Part of Complex Argument			
AIMAG(Z)	Reduce imaginary part of ordered pair to REAL*4	COMPLEX*8	REAL*4
DIMAG(Z)	Reduce imaginary part of ordered pair to REAL*8	COMPLEX*16	REAL*8
Conjugate of a Complex Argument			
CONJG(Z)	Conjugate of COMPLEX*8	COMPLEX*8	COMPLEX*8
DCONJG(Z)	Conjugate of COMPLEX*16	COMPLEX*16	COMPLEX*16
Square Root			
SQRT (generic)	Square root	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
SQRT	REAL*4 square root	REAL*4	REAL*4
DSQRT	REAL*8 square root	REAL*8	REAL*8
CSQRT	COMPLEX*8 square root	COMPLEX*8	COMPLEX*8
CDSQRT	COMPLEX*16 square root	COMPLEX*16	COMPLEX*16
Exponential			
EXP (generic)	Exponent	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
EXP(X)	REAL*4 e to power	REAL*4	REAL*4
DEXP(X)	REAL*8 e to power	REAL*8	REAL*8
CEXP	COMPLEX*8 e to power	COMPLEX*8	COMPLEX*8
CDEXP	COMPLEX*16 e to power	COMPLEX*16	COMPLEX*16

Table 5.1 *(continued)*

Name	Definition	Type of Argument	Type of Function
Natural Logarithm			
LOG (generic)	Natural logarithm	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
ALOG(X)	Nat'l log of REAL*4	REAL*4	REAL*4
DLOG(X)	Nat'l log of REAL*8	REAL*8	REAL*8
CLOG(Z)	Nat'l log of Complex	COMPLEX*8	COMPLEX*8
CDLOG(Z)	Nat'l log of Complex	COMPLEX*16	COMPLEX*16
Common Logarithm			
LOG10 (generic)	Common logarithm	REAL*4 REAL*8	REAL*4 REAL*8
ALOG10(X)	Common log of REAL*4	REAL*4	REAL*4
DLOG10(X)	Common log of REAL*8	REAL*8	REAL*8
Sine			
SIN (generic)	Sine function	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
SIN(X)	REAL*4 sine	REAL*4	REAL*4
DSIN(X)	REAL*8 sine	REAL*8	REAL*8
CSIN(Z)	COMPLEX*8 sine	COMPLEX*8	COMPLEX*8
CDSIN(Z)	COMPLEX*16 sine	COMPLEX*16	COMPLEX*16
Cosine			
COS (generic)	Cosine function	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
COS(X)	REAL*4 cosine	REAL*4	REAL*4
DCOS(X)	REAL*8 cosine	REAL*8	REAL*8
CCOS(Z)	COMPLEX*8 cosine	COMPLEX*8	COMPLEX*8
CDCOS(Z)	COMPLEX*16 cosine	COMPLEX*16	COMPLEX*16
Tangent			
TAN (generic)	Tangent function	REAL*4 REAL*8	REAL*4 REAL*8
TAN(X)	REAL*4 tangent	REAL*4	REAL*4
DTAN(X)	REAL*8 tangent	REAL*8	REAL*8

Table 5.1 *(continued)*

Name	Definition	Type of Argument	Type of Function
Arc Sine			
ASIN (generic)	Arc sine function	REAL*4 REAL*8	REAL*4 REAL*8
ASIN(X)	REAL*4 arc sine	REAL*4	REAL*4
DASIN(X)	REAL*8 arc sine	REAL*8	REAL*8
Arc Cosine			
ACOS (generic)	Arc cosine function	REAL*4 REAL*8	REAL*4 REAL*8
ACOS(X)	REAL*4 arc cosine	REAL*4	REAL*4
DACOS(X)	REAL*8 arc cosine	REAL*8	REAL*8
Arc Tangent			
ATAN (generic)	Arc tangent function	REAL*4 REAL*8	REAL*4 REAL*8
ATAN(X)	REAL*4 arc tangent	REAL*4	REAL*4
DATAN(X)	REAL*8 arc tangent	REAL*8	REAL*8
ATAN2 (generic)	Arctan (X/Y)	REAL*4 REAL*8	REAL*4 REAL*8
ATAN2(X,Y)	REAL*4 arctan of X/Y	REAL*4	REAL*4
DATAN2(X,Y)	REAL*8 arctan of X/Y	REAL*8	REAL*8
Hyperbolic Sine			
SINH (generic)	Hyperbolic sine	REAL*4 REAL*8	REAL*4 REAL*8
SINH(X)	REAL*4 hyperbolic sine	REAL*4	REAL*4
DSINH(X)	REAL*8 hyperbolic sine	REAL*8	REAL*8
Hyperbolic Cosine			
COSH (generic)	Hyperbolic cosine	REAL*4 REAL*8	REAL*4 REAL*8
COSH(X)	REAL*4 hyperbolic cosine	REAL*4	REAL*4
DCOSH(X)	REAL*8 hyperbolic cosine	REAL*8	REAL*8
Hyperbolic Tangent			
TANH (generic)	Hyperbolic tangent	REAL*4 REAL*8	REAL*4 REAL*8
TANH(X)	REAL*4 hyperbolic tangent	REAL*4	REAL*4
DTANH(X)	REAL*8 hyperbolic tangent	REAL*8	REAL*8

Table 5.1 *(continued)*

Name	Definition	Type of Argument	Type of Function
Lexically Greater Than or Equal			
LGE(C1,C2) [7]	1st argument greater than or equal to 2nd	CHARACTER	LOGICAL
Lexically Greater Than			
LGT(C1,C2) [7]	1st argument greater than 2nd	CHARACTER	LOGICAL
Lexically Less Than or Equal			
LLE(C1,C2) [7]	1st argument less than or equal to 2nd	CHARACTER	LOGICAL
Lexically Less Than			
LLT(C1,C2) [7]	1st argument less than 2nd	CHARACTER	LOGICAL
End of File [8]			
EOF(I)	INTEGER end of file	INTEGER	LOGICAL

Notes for Table 5.1:

1. For X of type INTEGER, INT(X)=X. For X of type REAL or REAL*8, if X is greater than or equal to zero, then INT(X) is the largest integer not greater than X, and if X is less than zero, then INT(X) is the most negative integer not less than X. For X of type REAL, IFIX(X) is the same as INT(X).
2. For X of type REAL, REAL(X)=X. For X of type INTEGER or REAL*8, REAL(X) is as much precision of the significant part of X as a real datum can contain. For X of type INTEGER, FLOAT(X) is the same as REAL(X).
3. For X of type REAL*8, DBLE(X)=X. For X of type INTEGER or REAL, DBLE(X) is as much precision of the significant part of X as a double precision datum can contain.

Notes for Table 5.1 (continued)

4. CMPLX and DCMPLX may have one or two arguments. If there is one argument, it may be of type integer, real, double precision, complex, or double precision complex. If there are two arguments, they must be of the same type and the allowable types are integer, real and double precision.

   Where Z=COMPLEX*8, CMPLX(Z)=Z. For Z= INTEGER, REAL, and REAL*8 then CMPLX(Z)= the complex value whose real part is REAL(Z) and whose imaginary part is 0.

   CMPLX(Z,Y)= the complex value whose real part is REAL(Z) and whose imaginary part is REAL(Y).

   Where Z=COMPLEX*16, DCMPLX(Z)=Z. For Z= INTEGER, REAL, and REAL*8 then DCMPLX(Z)= the complex value whose real part is REAL*8 and whose imaginary part is 0. For Z=COMPLEX*8, DCMPLX(Z) has a real part equal to DBLE(Z) and an imaginary part equal to DBLE(AIMAG(Z)).

   DCMPLX(Z,Y)= the complex value whose real part is REAL*8(Z) and whose imaginary part is REAL*8(Y).

5. ICHAR converts a character value into an integer value. The integer value of a character is the ASCII internal representation of that character, and is in the range 0 to 255. For any two characters, c1 and c2, (c1 .LE. c2) is .TRUE. if and only if (ICHAR(c1) .LE. ICHAR(c2)) is .TRUE.

   CHAR(*i*) returns the *i*th character in the collating sequence. The value is of type CHARACTER, length one, while *i* must be an integer expression whose value is in the range 0 <= *i* <= 255.

   ICHAR(CHAR(*i*)) = *i* for 0 <= *i* <= 255.

   CHAR(ICHAR(c)) = c for any character c in the character set.

6. DIM(X,Y) is X-Y if X>Y, zero otherwise.

7. LGE(X,Y) returns the value .TRUE. if X = Y or if X follows Y in the ASCII collating sequence; otherwise it returns .FALSE.

   LGT(X,Y) returns .TRUE. if X follows Y in the ASCII collating sequence; otherwise it returns .FALSE.

   LLE(X,Y) returns .TRUE. if X = Y or if X precedes Y in the ASCII collating sequence; otherwise it returns .FALSE.

   LLT(X,Y) returns .TRUE. if X precedes Y in the ASCII collating sequence; otherwise it returns .FALSE.

   If the operands are of unequal length, the shorter operand is considered to be blankfilled on the right to the length of the longer.

8. EOF(I) returns the value .TRUE. if the unit specified by its argument is at or past the end of file record; otherwise it returns .FALSE. The value of X must correspond to an open file. EOF cannot refer to Unit 0 (the screen or keyboard).

5.3.3 Statement Functions

A statement function is defined by a single statement and is similar in form to an assignment statement. A statement function statement can only appear after the specification statements and before any executable statements in the program unit in which it appears.

A statement function is not an executable statement, since it is not executed in order as the first statement in its particular program unit. Rather, the body of a statement function serves to define the meaning of the statement function. It is executed, as any other function, by the execution of a function reference in an expression.

For information on the syntax and use of a statement function statement, see Section 3.2.38, "The Statement Function Statement."

5.4 Arguments

A formal argument is the name by which the argument is known within a function or subroutine; an actual argument is the specific variable, expression, array, etc., passed to the procedure in question at any specific calling location. The relationship between formal and actual arguments in a function or subroutine call is discussed in detail in the following paragraphs.

Arguments pass values into and out of procedures by reference. The number of actual arguments must be the same as the number of formal arguments, and the corresponding types must agree.

Upon entry to a subroutine or function, the actual arguments are associated with the formal arguments, much as an EQUIVALENCE statement associates two or more arrays or variables, and COMMON statements in two or more program units associate lists of variables. This association remains in effect until execution of the subroutine or function is terminated. Thus, assigning a value to a formal argument during execution of a subroutine or function may alter the value of the corresponding actual argument.

If an actual argument is a constant, function reference, or an expression other than a simple variable, assigning a value to the corresponding formal argument is not permitted, and can have some strange side effects. In particular, assigning a value to a formal argument of type CHARACTER, when the actual argument is a literal, can produce anomalous behavior.

If an actual argument is an expression, it is evaluated just before the association of formal and actual arguments. If an actual argument is an array element, its subscript expressions are also evaluated just before the association, and remain constant throughout the execution of the procedure, even if they contain variables that are redefined during the execution of the procedure.

A formal argument that is a variable can be associated with an actual argument that is a variable, an array element, or an expression.

A formal argument that is an alternate return (*) can be associated with an alternate return specifier (*n) in the CALL statement and is repeatable.

A formal argument that is an array can be associated with an actual argument that is an array or an array element. The number and size of dimensions in a formal argument may be different from those of the actual argument, but any reference to the formal array must be within the limits of the memory sequence in the actual array. While a reference to an element outside these bounds is not detected as an error in a running MS-FORTRAN program, the results are unpredictable.

A formal argument may also be associated with an external subroutine, function, or intrinsic function if it is used in the body of the procedure as a subroutine or function reference, or if it appears in an EXTERNAL statement.

A corresponding actual argument must be an external subroutine or function, declared with the EXTERNAL statement, or an intrinsic function permitted to be associated with a formal procedure argument. The intrinsic function must have been declared with an INTRINSIC statement in the program unit where it is used as an actual argument.

All intrinsic functions, except the following, may be associated with formal procedure arguments:

INT	AMAX0	DMIN1
IFIX	AMAX1	LGE
IDINT	MAX	LGT
FLOAT	MAX0	LLE
SNGL	MAX1	LLT
REAL	MIN	
DBLE	MIN0	
CMPLX	MIN1	
DCMPLX	AMIN0	
ICHAR	AMIN1	
CHAR	DMAX1	

Chapter 6
The Microsoft FORTRAN Metacommands

6.1 Overview

Metacommands are directives that order the MS-FORTRAN Compiler to process MS-FORTRAN source text in a specific way. MS-FORTRAN metacommands are described briefly in Table 6.1 and discussed in more detail in the remainder of the chapter.

Table 6.1

The Microsoft FORTRAN Metacommands

Metacommand	Action
$DEBUG	Turns on runtime checking for integer arithmetic operations and assigned GOTO values. $NODEBUG turns checking off. $DEBUG doesn't trigger or suppress floating-point exceptions.
$DECMATH	Directs the compiler to generate real constants in decimal floating-point format.
$DO66	Causes DO statements to have FORTRAN 66 semantics.
$FLOATCALLS	Directs compiler to generate calls to subroutines in the emulator library. $NOFLOATCALLS causes the compiler to generate in-line interrupt instructions.
$INCLUDE:*'file'*	Directs compiler to proceed as if *file* were inserted at that point.
$LARGE [:name [, name]]	Labels the named array for addressing outside of the DGROUP. $NOTLARGE disables the 'large' function for the named array. If [name] is missing, these commands affect all arrays.
$LINESIZE:*n*	Makes subsequent pages of listing *n* columns wide.
$LIST	Sends subsequent listing information to the listing file. $NOLIST stops generation of listing information.
$MESSAGE:'quoted string'	Sends a quoted character string to the standard output device.
$PAGE	Starts new page of listing.
$PAGESIZE:*n*	Makes subsequent pages of listing *n* lines long, minimum *n*=15.
$STORAGE:*n*	Allocates *n* bytes of memory to all LOGICAL or INTEGER variables in source.

Table 6.1 *(continued)*

Metacommand	Action
$STRICT	Disables MS-FORTRAN features not in 1977 subset or full language standard. $NOTSTRICT enables them.
$SUBTITLE:'subtitle'	Gives subtitle for subsequent pages of listing.
$TITLE:'title'	Gives title for subsequent pages of listing.

Metacommands can be intermixed with MS-FORTRAN source text within an MS-FORTRAN source program; however, they are not part of the standard FORTRAN language. Any line of input to the MS-FORTRAN Compiler that begins with a "$" character in column one is interpreted as a metacommand and must conform to one of the allowable formats.

A metacommand and its arguments (if any) must fit on a single source line; continuation lines are not permitted. Also, blanks are significant, so that the following pair is not equivalent:

```
$S TRICT
$STRICT
```

6.2 Metacommand Directory

The remainder of this chapter is an alphabetical directory of available MS-FORTRAN metacommands.

6.2.1 The $DEBUG and $NODEBUG Metacommands

Syntax

$[NO]DEBUG

Purpose

Directs the compiler to (1) test integer arithmetic for overflow and division by zero, (2) test assigned GOTO values against the allowable list in an assigned GOTO statement, and (3) provide the runtime error-handling system with source filenames and line numbers. If any runtime error occurs, the filename and line number are displayed on the console.

Remarks

The metacommand can appear anywhere in a program. BUG does not trigger or suppress floating-point exception handling. MS-FORTRAN conforms to the proposed IEEE Standard for exception handling for the five following conditions: invalid operation, divide by zero, overflow, underflow, and precision. See the *Microsoft FORTRAN Compiler User's Guide* for information about exception handling on your system.

The default value of the pair of metacommands, $DEBUG and $NODEBUG, is $NODEBUG.

6.2.2 The $DECMATH Metacommand

Syntax

$DECMATH

Purpose

Directs the compiler (and the runtime for the compiled program) to have floating-point math performed in base-10 rather than binary. Specifically, DECMATH forces constants to be represented in base-10 math format.

Remarks

This metacommand must appear before the first program or subprogram statement; it can only be preceded by comment lines and other metacommands in the source and may only be given once.

$DECMATH automatically sets $FLOATCALLS. If $DECMATH is set, $FLOATCALLS and $NOFLOATCALLS will be ignored.

6.2.3 The $DO66 Metacommand

Syntax

$DO66

Purpose

Causes DO statements to have FORTRAN 66 semantics.

Remarks

$DO66 must precede the first declaration or executable statement of the source file in which it occurs. $DO66 may only be preceded by a comment line or another metacommand. $DO66 may only appear once in the source file.

The FORTRAN 66 semantics are as follows:

1. All DO statements are executed at least once.
2. Extended range is permitted; that is, control may transfer into the syntactic body of a DO statement. The range of the DO statement is thereby extended to logically include any statement that may be executed between a DO statement and its terminal statement. However, the transfer of control into the range of a DO statement prior to the execution of the DO statement or following the final execution of its terminal statement is invalid.

If a program contains no $DO66 metacommand, the default is to FORTRAN 77 semantics, as follows:

1. DO statements may be executed zero times, if the initial control variable value exceeds the final control variable value (or the corresponding condition for a DO statement with negative increment).
2. Extended range is invalid; that is, control may not transfer into the syntactic body of a DO statement. (Both standards do permit transfer of control out of the body of a DO statement.)

6.2.4 The $FLOATCALLS and $NOFLOATCALLS Metacommmands

Syntax

$[NO]FLOATCALLS

Purpose

The $FLOATCALLS metacommand causes your floating-point operations to be processed by calls to library subroutines.

Remarks

When you enter $FLOATCALLS into your source, call instructions are generated for real number computations. $NOFLOATCALLS suppresses the default condition and causes the compiler to generate in-line interrupt instructions rather than subroutine calls. $FLOATCALLS is the default.

$DECMATH automatically sets $FLOATCALLS; if $DECMATH is given, neither $FLOATCALLS nor $NOFLOATCALLS may appear. (See $DECMATH for more information).

6.2.5 The $INCLUDE Metacommand

Syntax

$INCLUDE: '*file*'

Purpose

Directs the compiler to proceed as though the specified file were inserted at the point of the $INCLUDE.

Remarks

file is a valid file specification as described for your operating system.

At the end of the included file, the compiler resumes processing the original source file at the line following $INCLUDE.

The compiler imposes no limit on nesting levels for $INCLUDE metacommands. $INCLUDE metacommands are particularly useful for guaranteeing that several modules use the same declaration for a COMMON block.

6.2.6 The $LARGE and $NOTLARGE Metacommands

Syntax

$[NOT]LARGE [: *name* [, *name*]....]

Purpose

The $LARGE metacommand is supplied for use on systems whose memory model makes programs more efficient if the compiler can assume that both the size of arrays and the total amount of non-COMMON data is restricted. $LARGE directs the compiler to allocate arrays in a less restricted way and to generate the less efficient code sequences to reference them. See the *Microsoft FORTRAN Compiler User's Guide* for details that apply to your system.

Remarks

$LARGE may be used without arguments, the "generic" case. This form may occur anywhere except in the executable section of a subprogram. $LARGE affects all subprograms that follow its occurrence in the source file unless a generic $NOTLARGE metacommand is issued in subsequent code. $NOTLARGE follows the same rules as $LARGE but has the opposite effect. $NOTLARGE is the default.

In the region between an END statement (or the start of the compilation unit) and the executable part of the next subprogram or main program, the generic form of $LARGE or $NOTLARGE but not both may occur only once. For example, the following code fragment is illegal;

```
$LARGE
        SUBPROGRAM P
$NOTLARGE
        A=1.0
        .
        .
```

$LARGE and $NOTLARGE can also take valid array variable names and formal array arguments. The array names must be followed by a colon. If arrays are specified with $LARGE, the metacommand must occur in the declarative section of a subprogram, and it only affects the arrays or formal array arguments declared in that subprogram.

Note

Arrays with explicit dimensions that indicate that they are bigger than the allowable limit of 64K bytes are automatically allocated to multiple segments outside of the default data segment. You do not need to issue $LARGE for these arrays.

6.2.7 The $LINESIZE Metacommand

Syntax

$LINESIZE:*n*

Purpose

Formats subsequent pages of the listing *n* columns wide.

Remarks

n is any positive integer.

If a program contains no $LINESIZE metacommand, a default line size of 80 characters is assumed. The minimum line size is 40 characters, the maximum is 132 characters.

6.2.8 The $LIST and $NOLIST Metacommands

Syntax

$[NO]LIST

Purpose

Sends subsequent listing information to the listing file specified when starting the compiler. If no listing file is specified in response to the compiler prompt, the metacommand has no effect. $NOLIST directs that subsequent listing information be discarded.

Remarks

$LIST and $NOLIST can appear anywhere in a source file.

The default condition for the pair of metacommands, $LIST and $NOLIST, is $LIST.

6.2.9 The $MESSAGE Metacommand

Syntax

$MESSAGE: ' *quoted string* '

Purpose

Instructs the compiler to display a quoted string at the standard output device during the compilation.

Remarks

The $MESSAGE metacommand can be used to send a quoted string to the standard output device when running MS-FORTRAN FOR1.EXE. For example, the entry

```
$MESSAGE 'Phase I being compiled'
```

will deliver the character string in single quotes to the output device.

The maximum length of the string is 40 characters.

6.2.10 The $PAGE Metacommand

Syntax

$PAGE

Purpose

Starts a new page of the listing.

Remarks

If the first character of a line of source text is the ASCII form feed character (hexadecimal code 0Ch), it is considered as equivalent to the occurrence of a $PAGE metacommand at that point.

6.2.11 The $PAGESIZE Metacommand

Syntax

$PAGESIZE:*n*

Purpose

Formats subsequent pages of the listing *n* lines high.

Remarks

n must be at least 15.

If a program contains no $PAGESIZE metacommand, a default page size of 66 lines is assumed.

6.2.12 The $STORAGE Metacommand

Syntax

$STORAGE:*n*

Purpose

Allocates *n* bytes of memory for all variables declared in the source file as INTEGER or LOGICAL.

Remarks

n is either 2 or 4. Use a value of 2 for code that defaults to 16-bit arithmetic. See also the important note on performance issues in Section 2.3, "Data Types."

$STORAGE does not affect the allocation of memory for variables declared with an explicit length specification, for example, as INTEGER*2 or LOGICAL*4.

If several files of a source program are compiled and linked together, you should be particularly careful that they are consistent in their allocation of memory for variables (such as actual and formal parameters) referred to in more than one module.

The $STORAGE metacommand must precede the first declaration statement of the source file in which it occurs.

If a program contains no $STORAGE metacommand, a default allocation of 4 bytes is used. This default results in INTEGER, LOGICAL, and REAL variables being allocated the same amount of memory, as required by the FORTRAN 77 standard.

6.2.13 The $STRICT and $NOTSTRICT Metacommands

Syntax

$[NOT]STRICT

Purpose

$STRICT disables the specific MS-FORTRAN features not found in the FORTRAN 77 subset or full language standard.

Remarks

The $NOTSTRICT metacommand enables these MS-FORTRAN features, which are the following:

1. Character expressions may be assigned to noncharacter variables.
2. Character and noncharacter expressions may be compared.
3. Character and noncharacter variables are allowed in the same COMMON block.
4. Character and noncharacter variables may be equivalenced.
5. Noncharacter variables may be initialized with character data.

$STRICT and $NOTSTRICT can appear anywhere in a source file.

The default condition for the pair of metacommands, $STRICT and $NOTSTRICT, is $NOTSTRICT.

6.2.14 The $SUBTITLE Metacommand

Syntax

$SUBTITLE: '*subtitle*'

Purpose

Assigns the specified subtitle for subsequent pages of the source listing (until overridden by another $SUBTITLE metacommand).

Remarks

subtitle is any valid character constant. The maximum length is 40 characters.

If a program contains no $SUBTITLE metacommand, the subtitle is a null string.

6.2.15 The $TITLE Metacommand

Syntax

$TITLE: *'title'*

Purpose

Assigns the specified title for subsequent pages of the listing (until overridden by another $TITLE metacommand).

Remarks

title is any valid character constant. The maximum length is 40 characters.

If a program contains no $TITLE metacommand, the title is a null string.

Appendices

Appendix A
Microsoft FORTRAN and ANSI Subset FORTRAN

This appendix describes how MS-FORTRAN differs from the standard subset language. The ANSI standard defines two levels, full FORTRAN and subset FORTRAN. MS-FORTRAN is a superset of the latter. The differences between MS-FORTRAN and the standard subset FORTRAN fall into two general categories: full language features and extensions to the standard.

A.1 Full Language Features

Several features from the full language are included in this implementation. In all cases, a program written to comply with the subset restrictions compiles and executes properly, since the full language includes the subset constructs.

1. Subscript expressions

   The subset does not allow function calls or array element references in subscript expressions; however, these are allowed in the full language and in this implementation.

2. DO variable expressions

   The subset restricts expressions that define the limits of a DO statement; the full language does not. MS-FORTRAN also allows full integer expressions in DO statement limit computations. Similarly, arbitrary integer expressions are allowed in implied DO loops associated with READ and WRITE statements.

3. Unit I/O number

   MS-FORTRAN allows an I/O unit to be specified by an integer expression, as does the full language.

4. Expressions in input/output list <iolist>

   The subset does not allow expressions to appear in an <iolist>, whereas the full language does allow expressions in the <iolist> of WRITE statements. MS-FORTRAN allows expressions in the <iolist> of a WRITE statement providing that the expressions do not begin with an initial left parenthesis.

   Note that an expression like (A+B)*(C+D) can be specified in an output list as +(A+B)*(C+D). Doing so does not generate any extra code to evaluate the leading plus sign.

5. Double precision and Complex numbers

   The subset does not allow double precision real numbers or complex numbers; MS-FORTRAN provides for them as in the full language standard.

6. Edit descriptors

   MS-FORTRAN allows for D, G, T, TR, TL, S, SS, SP, and (:) edit descriptors as in the full language standard.

7. Expression in computed GOTO

   MS-FORTRAN allows an expression for the selector of a computed GOTO, consistent with the full, rather than the subset, language.

8. Generalized I/O

   MS-FORTRAN allows both sequential and direct access files to be either formatted or unformatted. The subset language requires direct access files to be unformatted and sequential files to be formatted.

   MS-FORTRAN also includes the following:

   a. an augmented OPEN statement that takes additional parameters not included in the subset (see Section 3.2.28, "The OPEN Statement")

   b. a form of the CLOSE statement, which is not included in the subset (see Section 3.2.5, "The CLOSE Statement")

c. a form of the INQUIRE statement which is not included in the subset

d. END=, ERR=, IOSTAT=, STATUS=, and FILE= specifiers on I/O statements

9. List-directed I/O

   MS-FORTRAN provides for list-directed I/O as described in the full language standard.

10. Array dimensions

    MS-FORTRAN supports seven (7) array dimensions in conformance with the full language standard.

11. Continuation lines

    MS-FORTRAN permits 19 continuation line in conformance with the full language standard.

12. The BLOCK DATA statement

    MS-FORTRAN implements the BLOCK DATA statement but does not check for all of the restrictions that apply in the full language standard.

13. CHARACTER-typed function

    Subject to restriction, functions may be of type CHARACTER.

A.2 Extensions to the Standard

The implemented language also has several minor extensions to the full language standard.

1. User-defined names greater than six characters are allowed, although only the first six characters are significant.
2. Tabs in source files are allowed. See Section 2.1.3, "Tabs," for details.

3. Metacommands, or compiler directives, have been added to allow the programmer to communicate certain information to the compiler. The metacommand line is characterized by a dollar sign ($) appearing in column 1. A metacommand line may appear any place that a comment line can appear, although certain metacommands are restricted as to their location within a program (see Section 2.2.4, "Statement Definition and Order").

   A metacommand line conveys certain compile time information about the nature of the current compilation to the MS-FORTRAN Compiler. Metacommands are described in Chapter 6, "The Microsoft FORTRAN Metacommands."

4. The standard is relaxed when the $NOTSTRICT metacommand is in effect. This relaxation allows, for example, such MS-FORTRAN features as assignment of character to any variable type and initialization of any variable with character data. See Section 6.2.13, "The $STRICT and $NOTSTRICT Metacommands, for a complete list of these features.

5. The backslash (\) edit control character can be used in format specifications to inhibit normal advancement to the next record associated with the completion of a READ or WRITE statement. This is particularly useful when prompting to an interactive device, such as the screen, so that a response can appear on the same line as the prompt. See item 5, Section 4.4.2.1, "Nonrepeatable Edit Descriptors," for more information.

6. An end of file intrinsic function, EOF, is provided. The function accepts a unit specifier as an argument and returns a logical value that indicates whether the specified unit is at its end of file.

7. Both upper and lowercase source input are allowed. In most contexts, lowercase characters are treated as indistinguishable from their uppercase counterparts. However, lowercase is significant in character constants and Hollerith fields.

8. Binary files are similar to unformatted sequential files except that they have no internal structure. This allows the program to create or read files with arbitrary contents, which is particularly useful for files created by or intended for programs written in languages other than FORTRAN.

9. COMPLEX*16; MS-FORTRAN supports a 16 byte representation of a complex number (an ordered pair of DOUBLE PRECISION numbers).
10. If an input operation READs more characters from a formatted record than the record contains, the record is padded with blanks on the right.
11. When READing numeric and logical items using formatted input/output, the input fields may be delimited by commas, overriding the field width specification.

Appendix B
ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	036	24H	$
001	01H	SOH	037	25H	%
002	02H	STX	038	26H	&
003	03H	ETX	039	27H	'
004	04H	EOT	040	28H	(
005	04H	ENQ	041	29H	)
006	06H	ACK	042	2AH	*
007	07H	BEL	043	2BH	+
008	08H	BS	044	2CH	,
009	09H	HT	045	2DH	-
010	0AH	LF	046	2EH	.
011	0BH	VT	047	2FH	/
012	0CH	FF	048	30H	0
013	0DH	CR	049	31H	1
014	0EH	SO	050	32H	2
015	0FH	SI	051	33H	3
016	10H	DLE	052	34H	4
017	11H	DC1	053	35H	5
018	12H	DC2	054	36H	6
019	13H	DC3	055	37H	7
020	14H	DC4	056	38H	8
021	15H	NAK	057	39H	9
022	16H	SYN	058	3AH	:
023	17H	ETB	059	3BH	;
024	18H	CAN	060	3CH	<
025	19H	EM	061	3DH	=
026	1AH	SUB	062	3EH	>
027	1BH	ESCAPE	063	3FH	?
028	1CH	FS	064	40H	@
029	1DH	GS	065	41H	A
030	1EH	RS	066	42H	B
031	1FH	US	067	43H	C
032	20H	SPACE	068	44H	D
033	21H	!	069	45H	E
034	22H	"	070	46H	F
035	23H	#	071	47H	G

Dec	Hex	CHR	Dec	Hex	CHR
072	48H	H	101	65H	e
073	49H	I	102	66H	f
074	4AH	J	103	67H	g
075	4BH	K	104	68H	h
076	4CH	L	105	69H	i
077	4DH	M	106	6AH	j
078	4EH	N	107	6BH	k
079	4FH	O	108	6CH	l
080	50H	P	109	6DH	m
081	51H	Q	110	6EH	n
082	52H	R	111	6FH	o
083	53H	S	112	70H	p
084	54H	T	113	71H	q
085	55H	U	114	72H	r
086	56H	V	115	73H	s
087	57H	W	116	74H	t
088	58H	X	117	75H	u
089	59H	Y	118	76H	v
090	5AH	Z	119	77H	w
091	5BH	[	120	78H	x
092	5CH	\	121	79H	y
093	5DH	]	122	7AH	z
094	5EH	^	123	7BH	{
095	5FH	_	124	7CH	|
096	60H	`	125	7DH	}
097	61H	a	126	EH	~
098	62H	b	127	7FH	DEL
099	63H	c			
100	64H	d			

Dec=decimal, Hex=hexadecimal (H), CHR=character,
LF=Line Feed, FF=Form Feed, CR=Carriage Return,
DEL=Rub Out

Appendix C

Structure of External Microsoft FORTRAN Files

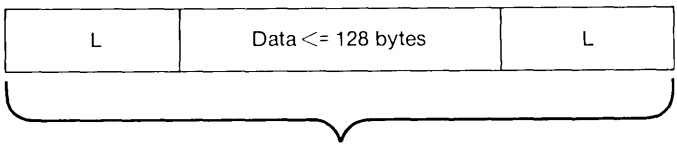
The structure of an external MS-FORTRAN file is determined by its properties. The structures used in MS-FORTRAN are as follows:

1. Formatted sequential files

   Records are separated by carriage return and linefeed (ASCII hex codes 0D and 0A, respectively).

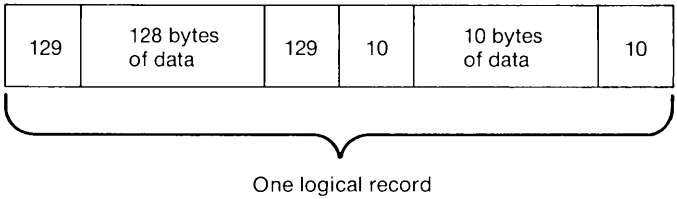
2. Unformatted sequential files

   A logical record is represented as a series of physical records, each of which has the following structure:



   Each *L* shown above is a length byte that indicates the length of the data portion of the physical record. The data portion of the last physical record contains MOD (length of logical record, 128) bytes, and the length bytes will contain the exact size of the data portion.

Each of the preceding physical records will contain 128 bytes in the data portion, while the length byte will contain 129. For example, if the size of the logical record is 138:



The first byte of the file is reserved and contains the value 75, which has no other significance.

3. Formatted direct files, unformatted direct files, and binary files

   No record boundaries or any other special characters are used.

Index

MICROSOFT®
10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem
(Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

MICROSOFT®