Microsoft® FORTRAN Optimizing Compiler

for the MS-DOS® Operating System

Language Reference

Microsoft Corporation

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

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document Number 410500018-400-R10-0187
Part Number 00036

Contents

Figures

Tables

Chapter 1
Introduction

1.1 Overview

This chapter introduces the information in the *Microsoft® FORTRAN Compiler Language Reference*, describes the notational conventions used in the manual, and tells how to learn more about FORTRAN.

For information on how to compile and link FORTRAN programs on your system, and a discussion of some of the Microsoft FORTRAN language extensions, see the *Microsoft FORTRAN Compiler User's Guide*. To find out how to use the Microsoft CodeView™ Window-Oriented Debugger to debug your programs, see the Microsoft CodeView manual.

1.2 About This Manual

The *Microsoft FORTRAN Compiler Language Reference* defines the FORTRAN language as implemented by the Microsoft FORTRAN Optimizing Compiler, Version 4.0. It is intended as a reference for programmers who have experience in the FORTRAN language. This manual does not teach you how to program in FORTRAN; for a list of suggested texts on FORTRAN, see Section 1.4, "Books about FORTRAN."

Microsoft FORTRAN conforms to the American National Standard Programming Language FORTRAN 77, as described in the American National Standards Institute (ANSI) X3.9-1978 standard.

Note

Microsoft FORTRAN contains many extensions to the full ANSI standard. In this manual, information on all Microsoft extensions is printed in blue.

Chapters 2, 3, and 4 of this manual discuss particular elements of the Microsoft FORTRAN language. Chapters 5 and 6 and Appendix B give alphabetical listings of statements, metacommands, and intrinsic functions, respectively. The following list shows where to look for information on specific topics:

For Information on:	See:
Characters, names, data types, attributes, and expressions in Microsoft FORTRAN	Chapter 2, "Elements of FORTRAN"
Formatting lines in your source program	Chapter 3, "Program Structure"
Structuring your Microsoft FORTRAN programs	Chapter 3, "Program Structure"
Subroutines, functions, and arguments	Chapter 3, "Program Structure"
Input and output in Microsoft FORTRAN	Chapter 4, "The Input/Output (I/O) System"
Microsoft FORTRAN statements, listed alphabetically	Chapter 5, "Statements"
Compiler directives, called metacommands, listed alphabetically	Chapter 6, "Metacommands"
Table of the American Standard Code for Information Interchange (ASCII) character set	Appendix A, "ASCII Character Codes"
Intrinsic functions, listed alphabetically	Appendix B, "Intrinsic Functions"
Selected terms used in this documentation	"Glossary"

1.3 Notational Conventions

This manual uses the following notation. Note that, in most cases, blanks are not significant in FORTRAN. See Section 2.2.1, "Blanks," for more information.

Example of Convention	Description of Convention
Extensions to ANSI standard	Blue type in this manual describes features that are extensions to the ANSI FORTRAN 77 full-language standard. These extensions may

or may not be implemented by other compilers that conform to the full-language standard.

For example, the sentence, "The **COMPLEX** or **COMPLEX*8** data type is an ordered pair of single-precision real numbers," has the words "or **COMPLEX*8**" in blue type, because the **COMPLEX*8** data type is an extension to the ANSI FORTRAN 77 full-language standard.

Examples

The typeface shown in the left column is used to simulate the appearance of information that would be printed on your screen or by your printer. For example, the following program is printed in this special typeface:

```
CHARACTER a*26
a = 'abcdefghijklmnopqrstuvwxyz'
WRITE(*,*)a(14:), a(:13)
END
```

When discussing this program in text, words appearing in the program, such as `a` and `WRITE`, also appear in the special typeface.

FORTRAN KEYWORDS

Bold capital letters indicate FORTRAN keywords. These keywords are a required part of the statement syntax, unless they are enclosed in double brackets, as explained below. In programs you write, you can enter these FORTRAN keywords in uppercase (capital) letters and/or lowercase letters.

In the following statement, **IF** and **THEN** are FORTRAN keywords:

IF (*expression*) **THEN**

other keywords

Bold lowercase letters indicate keywords of other languages.

In the sentence, "The value that is returned by **LOCNEAR** is equivalent to a **near** function or data pointer in Microsoft C or an **adr** type in Microsoft Pascal," the word **LOCNEAR** is a FORTRAN keyword, and the words **near** and **adr** are keywords of Microsoft C and Microsoft Pascal, respectively.

[(, * / =)]

Bold type indicates any punctuation or symbols (such as commas, parentheses, semicolons, hyphens, equal signs, and operators) that you must type exactly as shown.

For example, the syntax of the intrinsic function **CHAR** is shown as **CHAR**(*int*) because you must enter the word **CHAR**, followed by a left parenthesis, followed by an integer, followed by a right parenthesis. Note that since **CHAR** is a FORTRAN keyword, you can enter it in either uppercase or lowercase letters.

Apostrophes: ' ' '

An apostrophe is entered as a single right quotation mark ('), not a single left quotation mark (`). Note that in the typeface used in examples, such as `'string'`, apostrophes look like this: '.

placeholders

Words in italics are placeholders for types of information that you must supply. A file name is an example of this kind of information.

In the following statement, *label* is italicized to show that this is a general form for the **GOTO** statement:

GOTO *label*

In an actual program statement, the placeholder *label* must be replaced by a specific line label, as in the following example:

```
GOTO 150
```

Italics are also occasionally used in the text for emphasis.

[[*optional items*]]

Double square brackets surround anything that is optional.

The following statement, for example, shows that entering a *message* is optional in the **STOP** statement:

STOP [[*message*]]

Thus, either of the following **STOP** statements is acceptable:

```
STOP
STOP 'End of program'
```

Note

Double square brackets (⟦ ⟧) are a syntax convention used in this manual to indicate optional items. Single square brackets ([ ]) are punctuation that should be typed where shown.

{choice1 | choice2}

Braces and a vertical bar indicate that you have a choice between two or more items. Braces enclose the choices, and vertical bars separate the choices. You must choose one of the items unless all of the items are also enclosed in double square brackets.

For example, the INTERFACE statement has the following syntax:

INTERFACE TO *{function|subroutine}*

This statement indicates that you must enter either a *function* or a *subroutine* after the words INTERFACE TO.

Repeating elements...

Three dots following an item indicate that more items having the same form may be entered.

For example, this is the syntax of the intrinsic function **MAX**:

MAX (*genA*,*genB*⟦,*genC*⟧...)

The dots following ⟦,*genC*⟧ indicate that you can enter more than two arguments before the final right parenthesis, provided the arguments are separated by commas.

```
Program
.
.
.
Fragment
```

A column of dots in syntax lines and program examples shows that a portion of the program has been omitted.

For instance, in the following program fragment, only two lines are shown, and the lines in between are omitted:

```
CALL getnum(I,*10)
.
.
.
SUBROUTINE getnum(I,*)
```

KEY NAMES

Small capital letters are used for the names of keys and key sequences, which you must press. Examples include ENTER and CONTROL-C.

■ Example

The following example shows how this manual's notational conventions are used to indicate the syntax of the **EXTERNAL** statement:

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

This syntax listing shows that when using the **EXTERNAL** statement, you must first enter the word **EXTERNAL** followed by a *name* that you specify. Then, you can optionally enter a left bracket ([), followed by attributes (*attrs*) that you specify, followed by a right bracket (]). If you want to specify more *name*s, optionally followed by attributes (*attrs*), you must enter a comma, followed by a name, optionally followed by a left bracket, attributes, and a right bracket. Because the [,*name*[[*attrs*]]] sequence is followed by three dots (...), you can enter as many of those sequences (a comma, followed by a name, optionally followed by attributes in brackets) as you want.

1.4 Books about FORTRAN

The following books contain information on FORTRAN programming:

Agelhoff, Roy, and Richard Mojena. *Applied FORTRAN 77, Featuring Structured Programming*. Belmont, Calif.: Wadsworth, 1981.

Ashcroft, J., R. H. Eldridge, R. W. Paulson, and G. A. Wilson. *Programming with FORTRAN 77*. Dobbs Ferry, N.Y.: Sheridan House, Inc., 1981.

Friedman, Frank and E. Koffman. *Problem Solving and Structured Programming in FORTRAN*. 2d ed. Reading, Mass.: Addison-Wesley, 1981.

Kernighan, Brian W. and P. J. Plauger. *The Elements of Programming Style*. New York, N.Y.: McGraw-Hill, 1978.

Wagener, Jerrold L. *FORTRAN 77: Principles of Programming*. New York, N.Y.: John Wiley and Sons, Inc., 1980.

These books are listed for your convenience only. Microsoft Corporation does not endorse these books or recommend them over others on the same subject.

Chapter 2
Elements of FORTRAN

2.1 Introduction

A FORTRAN source file is composed of characters, names, and numbers. This chapter explains these elements and discusses how they can be combined in expressions.

2.2 Characters

Microsoft FORTRAN source files can contain any printable characters in the ASCII character set. The ASCII character set, listed in Appendix A, "ASCII Character Codes," includes the following:

- The 52 uppercase and lowercase alphabetic characters (A through Z and a through z). Microsoft FORTRAN also treats the dollar sign ($) as an alphabetic character. When names are being collated, the dollar sign immediately follows uppercase Z.

  The Microsoft FORTRAN Compiler interprets lowercase letters as uppercase letters in all contexts except character constants and Hollerith fields (see Section 4.8.1.2 for an explanation of Hollerith fields). In character constants and Hollerith fields, case is significant. For example, the statements `WRITE(*,*)` and `write(*,*)` are identical, but the character constants `'ijk'` and `'IjK'` are different.

  There is one exception to case sensitivity in character constants. Character constants that are specified with the FORTRAN statements listed in Chapter 5, "Statements," are not case sensitive unless the **$STRICT** metacommand is specified. In the **CLOSE** statement, for example, you can enter a character constant to specify whether to keep or delete a file. The syntax of this option is [,**STATUS**=*status*], and the acceptable values of *status* are **'KEEP'** and **'DELETE'**. As long as the **$STRICT** metacommand is not set, setting **STATUS** equal to **'KEEP'** is equivalent to setting **STATUS** equal to **'keep'** or **'KeEp'**. Note that since **STATUS** is a FORTRAN keyword, it can be spelled in uppercase and/or lowercase letters.

- The 10 digits (0 through 9).

- All other printable characters in the ASCII character set

  ! # % & ' ( ) * + , - . / : ;

  < = > ? @ [ ] ^ _ ` { | } ~

- The blank character.
- The tab character.

The collating sequence for the Microsoft FORTRAN character set is the ASCII sequence.

2.2.1 Blanks

The blank character has no significance in a Microsoft FORTRAN source program, except as listed below, so you can use blanks to make your programs easier to read. The exceptions are the following:

- Blanks in character constants or Hollerith fields are significant.
- A blank or 0 in column 6 indicates an initial line (see Section 3.2, "Lines," for an explanation of initial lines).
- When the **BZ** edit descriptor is in effect, blanks in numeric input fields (other than leading blanks) are interpreted as zeros. You can change this with the **BN** and **BZ** edit descriptors (described in Section 4.8.1.10, "Blank Interpretation"), or with the **BLANK** keyword in the **OPEN** statement (described in Section 5.3.38, "The OPEN Statement").

2.2.2 Tabs

Interpretation of the tab character depends on which column the tab character is in:

Column	Interpretation
1–5	The character following the tab character in the source line is interpreted as being in column 7.
6–72	The tab character is interpreted as one blank, unless it is in a character or Hollerith constant (described in Section 4.8.1.2). A tab character in a character or Hollerith constant is interpreted as a tab character.

2.3 Names

Variables, arrays, functions, subprograms, and your program are identified by names. Microsoft FORTRAN defines some names; you define others. A name is a sequence of alphanumeric characters and must obey the following rules:

- The first character in a name must be alphabetic; the rest of the characters must be alphanumeric. Note that Microsoft FORTRAN allows the dollar sign as an alphabetic character that follows Z in the collating sequence for names.
- Blanks are ignored. Thus, variable names like `low voltage` and `lowvoltage` are identical variables to the compiler.
- Unless the $NOTRUNCATE metacommand is set, only the first six alphanumeric characters are significant and the rest are ignored. Blank characters do not count: the names `delcate` and `d e l  c a t  e` are both interpreted as `delcat` unless the $NOTRUNCATE metacommand is set.
- The compiler limits names to 31 characters. Your operating system or linker may impose other limits on name lengths. See the *Microsoft FORTRAN Compiler User's Guide* for information specific to your system.

Keywords are not reserved names as in other languages. The compiler recognizes keywords by their context. For example, a program can have an array named `IF`, `read`, or `Goto`. Using these names, however, can make programs harder to read and understand. For readability, programmers should avoid using names that look like parts of FORTRAN statements. Consider the following two statements:

```
DO 5 INC = 1.20
DO 5 INC = 1,20
```

The first statement assigns the value `1.20` to a variable named `DO5INC`. The second statement is the first statement of a **DO** loop. Note that the only difference between the two statements is that the first contains a period and the second contains a comma.

The following three predefined names cannot be used:

1. **_main**, which is the external name for main programs. (The use of "main" is permitted under certain conditions, but is not recommended. See Section 3.8, "Main Program," for more information.)
2. **COMMQQ**, which is the name for blank common blocks.
3. **BLKDQQ**, which is the default name for block-data subprograms.

Also, all names beginning with two underscore characters (`__`) or ending with **QQ**, such as `__main` or `MAINQQ`, are reserved by the compiler. If you need to use a name beginning with two underscore characters or ending with **QQ**, use the **ALIAS** attribute (described in Section 2.6.1).

2.3.1 Global and Local Names

There are two basic types of names:

Type	Description
Global names	Global names are recognized anywhere in a given program, so they can have only one global definition anywhere in that program. All subroutine, function, common-block, and program names are global. For example, if you use a subroutine named `Sort` in one program, you cannot also name a function `Sort` in that program. You can, however, use the name `Sort` as a local name (described below) in a different program unit, provided you do not reference the global name `Sort` within that unit. For example, a program containing a function named `Sort` can also contain a subroutine that declares a variable named `Sort`, as long as the subroutine does not call the function `Sort`. Common-block names are a special case of global names. You can use the same name for a common block in one program and for a local name in the same program. This is permitted because common-block names are always enclosed in slashes, and can therefore be

distinguished from other names. For example, if your program includes a common block named `/distance/`, you can also name an array in that program `distance` (arrays have local names).

Local names

Local names are defined only in a single program unit. In another program unit of the same program, the same name can be defined again, with the same definition or a different definition.

All variables, arrays, arguments, and statement functions have local names.

Arguments to statement functions are a special case of local names. These arguments are defined only in the statement-function statement. If, however, the arguments' names are used outside of the statement-function statement, the local variables in the enclosing subprogram must have the same data type as statement-function arguments with the same name. See Section 5.3.48, "The Statement-Function Statement," for more information.

2.3.2 Undeclared Names

If a name is not explicitly defined, the compiler classifies the name according to the context in which it is first encountered. If you have specified the $DECLARE metacommand, a warning message is generated at the first use of any variable that has not been declared in a specification statement. The following list explains how undeclared names are classified:

Use of Name	Classification
As a variable, or in a function call	The type of the variable or of the function return value is determined by the first letter of the name. By default, variables with names starting with the letters I, J, K, L, M, or N (uppercase or lowercase) are given the type **INTEGER**, while variables with names starting with any other letter or with a dollar sign are given the type **REAL.** You can use the

	IMPLICIT statement to change the association between type and first alphabetic character (including the dollar sign). For more information, see Section 5.3.31, "The IMPLICIT Statement."
As the target of a **CALL** statement	The compiler assumes the name is a subroutine name. If the definition of the subroutine precedes a **CALL** statement in the same source file that references that subroutine, the compiler checks that the number and type of the actual arguments in the **CALL** statement are consistent with those specified in the corresponding **SUBROUTINE** statement. Note also that the INTERFACE statement (described in Section 5.3.34) can be used to ensure that subprogram calls have the correct number and type of arguments.
In a function reference	The compiler assumes the name is a function name. If the definition of the function precedes a function reference in the same source file that references that function, the compiler checks that the number and type of the actual arguments in the function reference are consistent with those specified in the corresponding **FUNCTION** statement. Note also that the INTERFACE statement (described in Section 5.3.34) can be used to ensure that subprogram calls have the correct number and type of arguments.

2.4 Data Types

There are five basic types of data in Microsoft FORTRAN:

1. Integer (**INTEGER**, INTEGER*1, INTEGER*2, and INTEGER*4)
2. Real (**REAL**, REAL*4, **DOUBLE PRECISION**, or REAL*8)
3. Complex (**COMPLEX**, COMPLEX*8, and COMPLEX*16)
4. Logical (**LOGICAL**, LOGICAL*1, LOGICAL*2, and LOGICAL*4)
5. Character (**CHARACTER**[*n*], where 1 <= *n* <= 32,767)

The data type of a variable, an array, an array element, a constant with a symbolic name, or a function can be declared in a specification statement. If the data type has not been declared, the compiler determines the data type of a name by its first letter (as described in Section 2.3.2, "Undeclared Names"). A type statement can also include dimension information and can be used to initialize variables and arrays. See Chapter 5, "Statements," for detailed descriptions of type statements.

The following sections (Sections 2.4.1 – 2.4.6) describe each data type. Memory requirements are shown in Table 2.1.

Table 2.1
Memory Requirements

Type	Bytes	Notes
INTEGER	2 or 4	Defaults to 4 bytes. The setting of the **$STORAGE** metacommand determines the size of **INTEGER** and **LOGICAL** values.
INTEGER*1	1	
INTEGER*2	2	
INTEGER*4	4	
REAL	4	Same as REAL*4.
REAL*4	4	
DOUBLE PRECISION	8	Same as REAL*8.
REAL*8	8	
COMPLEX	8	Same as COMPLEX*8.
COMPLEX*8	8	
COMPLEX*16	16	
LOGICAL	2 or 4	Defaults to 4 bytes. The setting of the **$STORAGE** metacommand determines the size of **INTEGER** and **LOGICAL** values.
LOGICAL*1	1	
LOGICAL*2	2	
LOGICAL*4	4	
CHARACTER	1	**CHARACTER** and **CHARACTER*1** are the same.
CHARACTER* *n*	*n*	Maximum *n* is 32,767.

2.4.1 Integer Data Types

The integer data type is a subset of the integers. An integer value is an exact representation of the corresponding integer. Table 2.2 shows the different types of integers, how many bytes of memory each type occupies, and the range of each type. Note that variables and functions declared as **INTEGER** are allocated as **INTEGER*4**, unless the **$STORAGE** metacommand is used to specify a 2-byte memory allocation. The **$STORAGE** metacommand also determines the default storage size of integer constants. If the **$STORAGE:2** metacommand is specified, for example, integer constants are 2 bytes long, by default. If, however, a constant is outside the **INTEGER*2** range, that constant is given 4 bytes of storage.

Table 2.2
Integers

Data Type	Bytes	Range
INTEGER*1	1	−127 to 127
INTEGER*2	2	−32,767 to 32,767
INTEGER*4	4	−2,147,483,647 to 2,147,483,647
INTEGER	2 or 4	Depends on setting of **$STORAGE**

Note that the range of values does not include the highest negative number that could be represented in the given number of bytes. The number −128 is out of the range of the **INTEGER*1** data type, −32,768 is out of the range of the **INTEGER*2** data type, and −2,147,483,648 is out of the range of the **INTEGER*4** data type. These numbers are treated as undefined numbers, and may be used by the compiler for error checking.

■ Syntax

By default, constants are interpreted in base 10. To specify a constant that is not in base 10, use the following syntax:

[*sign*][[*base*]#]*constant*

The *sign* is an optional plus or minus sign. The *base* can be any integer from 2 through 36. If *base* is omitted but # is specified, the integer is interpreted in base 16. If both *base* and # are omitted, the integer is interpreted in base 10. For bases 11 through 36, the letters A through Z represent the base-10 numbers greater than 10. For base 36, for example, A represents 10, B represents 11, C represents 12, and so on, through Z, which represents 35. Note that the case of the letters is not significant.

■ Example

For example, the following seven integers are all assigned a value equal to 3,994,575 decimal:

```
L = 3994575
```

A decimal point is not allowed in an integer constant.

Integer constants must also be in the ranges specified above. However, for numbers with a radix other than 10, the compiler reads out-of-range numbers up to $2^{32}-1$. They are interpreted as the negative numbers with the corresponding internal representation. For example, `16#FFFFFFFF` results in an arithmetic value of −1.

2.4.2 The Single-Precision IEEE Real Data Type

The single-precision IEEE (Institute of Electrical and Electronics Engineers, Inc.) real data type (**REAL** or REAL*4) is a subset of the real numbers. A single-precision real value is normally an approximation of the real number desired and occupies 4 bytes of memory. The precision of this data type is between six and seven decimal digits. Note that you can specify more than six digits, but for a single-precision real number, only the first six decimal digits are significant. The range of single-precision real values includes the negative numbers from approximately −3.4028235E+38 to −1.1754944E−38, the number 0, and the positive numbers from approximately +1.1754944E−38 to +3.4028235E+38.

■ Syntax

The form of a real constant is as follows:

[[*sign*]][[*integer*]].[[*fraction*]][[**E***exponent*]]

Parameter	Value
sign	A sign (+ or −).
integer	An integer. Either *integer* or *fraction* may be omitted, but not both.
.	A decimal point.
fraction	A fraction part, consisting of one or more decimal digits. Either *fraction* or *integer* may be omitted, but not both.
E*exponent*	An exponent part, consisting of an optionally signed one- or two-digit integer constant. An exponent indicates that the value preceding the exponent is to be multiplied by ten raised to the value *exponent*.

■ Example

The following real constants all represent the same real number (one and twenty-three one-hundredths):

```
+1.2300E0       .012300E2       1.23E0          123E-2
+1.2300         123.0E-2        .000123E+4      1230E-3
```

2.4.3 The Double-Precision IEEE Real Data Type

The double-precision real data type (REAL*8 or **DOUBLE PRECISION**) is a subset of the real numbers. This subset is larger than the subset for the single-precision real data type. A double-precision real value is normally an approximation of the real number desired, and occupies 8 bytes of memory. The precision is greater than 15 decimal digits. Note that you can specify more digits, but only the first 15 are significant. The range of double-precision real values includes the negative numbers from approximately −1.797693134862316D+308 to −2.225073858507201D−308, and the number 0. It also includes the positive numbers from approximately +2.225073858507201D−308 to +1.797693134862316D+308.

A double-precision real constant has the same form as a single-precision real constant, except that the letter **D** is used for exponents instead of the letter **E**, and an exponent part is mandatory. If the exponent is omitted, the number is interpreted as a single-precision constant. The following double-precision real constants all represent fifty-two one-thousandths:

```
5.2D-2    +.00052D+2    .052D0    52.000D-3    52D-3
```

Note that a constant such as `.052` is treated as a single-precision value, because no exponent is specified.

2.4.4 Complex Data Types

The **COMPLEX** or **COMPLEX*8** data type is an ordered pair of single-precision real numbers. The **COMPLEX*16** data type is an ordered pair of double-precision real numbers. The first number in the pair represents the real part of a complex number, and the second number in the pair represents the imaginary part. Both the real and imaginary components of a **COMPLEX** or **COMPLEX*8** number are **REAL*4** numbers, so **COMPLEX** or **COMPLEX*8** numbers occupy 8 bytes of memory. Both the real and imaginary components of a **COMPLEX*16** number are **REAL*8** numbers, so **COMPLEX*16** numbers occupy 16 bytes of memory.

■ Syntax

[*sign*](*real*,*imag*)

Parameter	Value
sign	A sign (+ or −). If specified, the *sign* applies to both *real* and *imag*.
real	An integer or real number, representing the real part.
imag	An integer or real number, representing the imaginary part.

For example, the complex number `(7,3.2)` represents the number 7.0+3.2i. The number `-(-.11E2,#5F)` represents the number 11.0−95.0i.

2.4.5 Logical Data Types

The logical data type includes the two logical values, **.TRUE.** and **.FALSE.**. A **LOGICAL** variable occupies 2 or 4 bytes of memory, depending on the setting of the **$STORAGE** metacommand. The default is 4 bytes. The significance of a logical variable is unaffected by the **$STORAGE** metacommand, which functions primarily to allow compatibility with the ANSI requirement that logical, single-precision real, and integer variables are all the same size.

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

2.4.6 The Character Data Type

A character value is a sequence of one or more of the printable ASCII characters enclosed by a pair of apostrophes (').

Note

An apostrophe is entered as a single right quotation mark (’), not a single left quotation mark (‘). Note that in the typeface used in examples, such as `'string'`, apostrophes look like this: **'**.

The apostrophes that enclose the string are not stored with the string. To represent an apostrophe within a string, specify two consecutive apostrophes with no blanks between them. Blank characters and tab characters are permitted in character constants and are significant. The case of alphabetic characters is significant. The string can contain any printable characters in the ASCII character set. You can use C strings (as described in Section 2.4.6.1) to define strings with nonprintable characters, or to specify the null string.

The length of a character value is equal to the number of characters between the apostrophes. A pair of apostrophes counts as a single character. The length of a character variable, character array element, character function, or character constant with a symbolic name must be between 1 and 32,767. The length can be specified by any of the following:

- An unsigned integer constant in the range 1 – 32,767
- An integer constant expression in parentheses
- An asterisk in parentheses: (*)

See Section 5.3.6, "The CHARACTER Statement," for more information.

Some sample character constants are listed below:

String	Constant
`'String'`	`String`
`'1234!@#$'`	`1234!@#$`
`'Blanks count'`	`Blanks count`
`''''''`	`''`
`'Case Is Significant'`	`Case Is Significant`
`'"Double" quotes count'`	`"Double" quotes count`

Note that FORTRAN source lines are 72 characters long (characters in columns 73 – 80 are ignored by the compiler), and lines with less than 72 characters are padded with blanks. Therefore, when a character constant extends across a line boundary, its value includes any blanks added to lines. For example, look at the following FORTRAN statement:

```
      heading (secondcolumn) =  'Acceleration of Particles
     $from Group A'
```

That statement sets the array element `heading (secondcolumn)` to `'Acceleration of Particles from Group A'`. There are 14 blanks between `Particles` and `from` because the word `Particles` ends in column 58 of the assignment statement, and 14 blanks are added to pad this line.

When allocated in a common block, a character variable occupies 1 byte of memory for each character in the sequence. Character variables are assigned to contiguous bytes, independent of word boundaries. However, when character and noncharacter variables are allocated in the same common block, the compiler assumes that noncharacter variables following character variables always start on word boundaries. See Section 5.3.8, "The COMMON Statement," for more information on character variables in common blocks.

2.4.6.1 C Strings

String values in the C language are terminated with null characters (CHAR(0)), and may contain nonprintable characters (such as the new line and backspace). These can be specified using the backslash character as an escape character, followed by a single character indicating the nonprintable character desired. This type of string can be specified in Microsoft FORTRAN by using a standard string constant followed by the character **C**. The standard string constant is then interpreted as a C-language constant. Backslashes are treated as escapes, and a null character is automatically appended to the end of the string (even if the string already ends in a null character). Table 2.3 shows the valid escape sequences. If a string contains an escape sequence that isn't in the table (such as \Z), the backslash is ignored.

Table 2.3

C String Escape Sequences

Sequence	Character
\n	New line
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\'	Single quote
\"	Double quote
\\	Backslash
ooo	Octal bit pattern
\x*hh*	Hexadecimal bit pattern
\a	Bell

* In FORTRAN you must enter \' ' to indicate this escape sequence.

The string must be a valid FORTRAN string, as described in Section 2.4.6, "The Character Data Type." Note that because the C string escape sequences are initially treated as FORTRAN strings, all quotation marks in the string itself must be double quotation marks (' ') produced by striking the single quote (apostrophe) key twice. The escape sequence \' 'a produces

a syntax error because FORTRAN interprets the quotation mark as the end of a string. The correct form is `\''a`. C strings and ordinary strings differ only in how you specify the value of the string. The compiler, for example, treats both string types the same when padding or truncating a string on assignment.

The sequences *ooo* and \x*hh* allow any character in the ASCII character set to be given as a one- to three-digit octal or a one- to two-digit hexadecimal character code. The *o* digit must be in the range 0 – 7, and the *h* digit must be in the range 0 – F. For example, the C strings `'\010'C` and `'\x08'C` both represent a backspace character followed by a null character.

The C string `'\\abcd'C` is equivalent to the string `'\abcd'`, with a null character appended to the end. The string `"C` represents the ASCII null character. Note that the character constant `''` is illegal because it has a length of 0, but `"C` is legal because it has a length of 1.

2.4.6.2 Character Substrings

Substrings are of type character, and are used to access a contiguous part of a character variable.

■ Syntax

variable ([[*first*]:[[*last*]])

Parameter	Description
variable	A character variable or an element in a character array.
first	An arithmetic expression that defines the first (leftmost) character in the substring. The compiler truncates *first* to an integer value. The default for *first* is 1, so if *first* is unspecified, the substring starts with the first character in the string.
last	An arithmetic expression that defines the last (rightmost) character in the substring. The compiler truncates *last* to an integer value. The default for *last* is the length of the string, so if *last* is unspecified, the substring ends with the last character in the string.

If the $STRICT metacommand is present, the use of a noninteger expression for the *first* and *last* parameters causes an error. Note that *variable*(:) is equivalent to *variable*. The length of the substring is *last* − *first* + 1. For example, for a 10-byte character variable named `name` that contains the string `'Jane Doe '`, `name(:5)` is `'Jane '`, `name(5+1:)` is `'Doe '`, and `name(:)` is `'Jane Doe '`.

Warning

Where *length* is the length of the character variable, the following relationships must be true:

- *first* <= *last*

  For the 10-byte character variable `name`, for example, `name(6:5)` is not allowed.

- 1 <= *first* <= *length*

  For example, `name(0:4)` and `name(11:12)` are not allowed.

- 1 <= *last* <= *length*

  For example, `name(:0)` and `name(:11)` are not allowed.

If the $DEBUG metacommand is on, an error is generated if these relationships are not true. If $DEBUG is not on, the results are undefined.

■ Example

```
C This program writes the second half of
C the alphabet, followed by the first half.
      CHARACTER alpha*26
      alpha='abcdefghijklmnopqrstuvwxyz'
      WRITE(*,*)alpha(14:),alpha(:13)
      END
```

2.5 Arrays

The number of elements in an array is limited only by available memory. If, however, the $STRICT metacommand is specified, a warning is generated if more than seven dimensions are used. To reference an element of an array, use the following syntax:

■ Syntax

array (*subscripts*)

Parameter	Value
array	The name of the array. If the type of the array is not declared in a type statement, the array elements have the type indicated by the first letter of *array*.
subscripts	Subscript expression(s). If there is more than one subscript expression, they must be separated by commas. The number of subscript expressions must be the same as the number of dimensions the array was declared with. For information on declaring arrays, see Section 5.3.12, "The DIMENSION Statement." Each subscript must be an arithmetic expression. The result of the expression is converted to an integer by truncation. If the $STRICT metacommand is specified, each subscript must be an integer expression. Function references and array-element references are allowed. The value of *subscripts* can be positive, negative, or 0.

■ Examples

```
C Examples of array-element references
      DIMENSION A(3,2), B(0:2,0:3), C(4,5), D(5,6),
     +V(10),Q(3.2,4.5)
      EQUIVALENCE (X,V(1)),(Y,V(2))
      D(I,J) = D(I+B(I,J),J)/PIVOT
      C(I+1,J) = C(I,J) + A(I**2,K) * B(K*J,J-24)
      READ(*,*)(V(N),N=1,10)
```

2.6 Attributes

Attributes allow additional information to be specified about a variable, variable type, subprogram, or subprogram formal argument. Attributes allow you, for example, to use the calling conventions of Microsoft C or Pascal, pass arguments by value or by reference, use segmented or unsegmented addresses, specify that a formal argument can span more than one segment, or specify an external name for a subprogram or common block. Attributes can be used in subroutine and function definitions, after type declarations, and with the **INTERFACE** (see Section 5.2.34) and **ENTRY** (see Section 5.2.20) statements. Table 2.4 summarizes which attributes can be used with different objects. See Sections 2.6.1 - 2.6.10 for information on each attribute.

Table 2.4

Objects to Which Attributes Can Refer

Attribute	Variable and Array Declarations	Common Block Names	Subprogram Specifications and EXTERNAL Statements
ALIAS	Yes	Yes	Yes
C	Yes	Yes	Yes
EXTERN	Yes	No	No
FAR	Yes	Yes	Yes[1]
HUGE	Yes	Yes	No
NEAR	Yes	Yes	Yes[1]
PASCAL	Yes	Yes	Yes
REFERENCE	Yes	No	No
VALUE	Yes	No	No
VARYING	No	No	Yes

[1] FAR and NEAR cannot be used in ENTRY statements.

■ Syntax

[[*attrs*]]

Attributes follow the object to which they refer. If more than one attribute is specified, they must be separated by commas.

■ Examples

```
C In the following example, the integer x is passed by
C reference, using a short address (offset only).

      INTEGER x[REFERENCE,NEAR]

C In the following example, f is a Pascal function with
C arguments i, j, and k that are C integers.
C It also returns a C integer.

      INTERFACE TO INTEGER [c] FUNCTION f [PASCAL] (i,j,k)
      INTEGER [c] i,j,k
      END
```

2.6.1 ALIAS

This attribute allows you to specify an external name for a subprogram or common block. The name may differ from the name used in the declaration.

■ Syntax

ALIAS: *string*

Parameter	Description
string	A character constant (can be a C string, as described in Section 2.4.6.1). No transformations are performed on *string*. Lowercase letters, for example, are not converted to uppercase. This is useful when interfacing with case-sensitive languages, such as C.

Only the name specified in the subprogram's declaration is available for referring to the subprogram inside the same source file. Only the name is available for referring to the subprogram outside the source file.

You can also use the **ALIAS** attribute on an **INTERFACE** statement to redefine the name of a subprogram in another source file that you wish to call.

The **ALIAS** attribute cannot be used on formal arguments.

■ Example

```
C This SUBROUTINE statement gives the subroutine f the
C name OtherNameForF outside of this source file.

      SUBROUTINE f[ALIAS:'OtherNameForF']
```

2.6.2 C

The **C** attribute can be applied to subprograms, common blocks, and types. When applied to a subprogram, the **C** attribute defines the subprogram as having the same calling conventions as a Microsoft C procedure. The following list indicates the differences between the FORTRAN Pascal calling conventions and the C calling conventions:

Difference	Explanation
The order in which parameters are pushed on the stack	Microsoft FORTRAN and Pascal push parameters on the stack in the order in which they appear in the procedure declaration. Microsoft C, by default, pushes its parameters in the reverse order (to allow varying numbers of arguments).
The location of code that restores the stack when a procedure returns	In Microsoft FORTRAN and Pascal, this code is in the called procedure. This produces less code than Microsoft C, in which this code follows the procedure call.

Arguments to subprograms with the **C** attribute are passed by value unless the **REFERENCE** attribute is specified on the formal argument. (Note that the **VARYING** attribute can be specified only for subprograms that also have the **C** attribute.) The names of subprograms using the **C** attribute are modified automatically to make it easier to match naming conventions used in C. External names are changed to lowercase, and begin with an underscore (_). To use a name containing uppercase letters, use the **ALIAS** attribute (described in Section 2.6.1).

When the C attribute is applied to the INTEGER type, that type is a C integer. The default size for C and FORTRAN integers may or may not be the same, depending on which processor is being used. For example, on the 8086 processor, Microsoft FORTRAN assumes 32-bit integers by default, while C assumes 16 bit. On other machines, however, both languages may assume 32-bit integers. So when you compile your program for a particular processor, you can use the C attribute on integer variables you want to pass between FORTRAN and C to make sure the compiler uses the right size.

The C attribute cannot be used on formal arguments, except when the attribute is applied to the INTEGER type, as in the following:

```
INTEGER(C) argument
```

2.6.3 EXTERN

The EXTERN attribute can be used in variable declarations. It indicates that the variable is allocated in another source file. EXTERN must be used when accessing variables declared in other languages, and must not be used on formal arguments.

2.6.4 FAR

The FAR attribute, when used with formal arguments, specifies that the argument is to be passed using a segmented address. When used with variables, it specifies that the variable is allocated in far data areas.

2.6.5 HUGE

The HUGE attribute is a convenient way to specify that a formal argument may span more than one segment. The $LARGE metacommand specifies the same thing. The following two program fragments, for example, are identical:

```
      FUNCTION F(A[HUGE])
      DIMENSION A(200)

      $LARGE: A
      FUNCTION F(A)
      DIMENSION A(200)
```

The compiler does not ensure that HUGE is specified for all arguments that span more than one segment. Versions 3.3 and earlier of Microsoft Pascal

and Versions 5.0 and earlier of Microsoft C do not support parameters with HUGE attributes.

2.6.6 NEAR

The NEAR attribute specifies that the actual argument is in the default data segment and that only its offset is passed to the subprogram. To pass a variable parameter to Microsoft Pascal, specify both the REFERENCE and the NEAR attributes.

This attribute can also be used with common blocks. Common blocks having the NEAR attribute are mapped into the default data segment. The following syntax is used:

COMMON [/[*name*]/]*clist*[,][NEAR][/]...

The parameter *name* is the name of the common block. When no *name* is specified, all blank common blocks are put in the default data segment. NEAR must be specified for at least the first definition of the common block in the source file. You can, however, specify NEAR for any of the COMMON statements in a subprogram.

To make a common block near, specifying NEAR for all definitions of the common block is good programming practice. If, however, you are modifying an existing program, it can be easier to add a subroutine at the beginning of your source file to make common blocks near in the remainder of the program.

The advantage of putting common blocks in the default data segment is that you can specify addresses with offsets only. This generates smaller, more efficient code. If you do not specify NEAR, the compiler uses segmented addresses to refer to everything in common blocks.

If a common block is specified as near in one compiland, but not in another, it will be mapped into the default data segment. The compiland which recognizes it as near will use short addresses, and the other compiland will use long addresses. While this practice is not recommended, it does provide compatibility with libraries compiled with Version 3.2 of the compiler.

Actual arguments passed to a near formal argument must be in the default data segment. You cannot pass any of the following to a near argument.

- Data in common blocks that are not specified as **NEAR**
- Arrays specified as **HUGE**
- Arrays defined while the **$LARGE** metacommand is in effect
- Variables named in a **$LARGE** metacommand

2.6.7 PASCAL

The **PASCAL** attribute can be used with subprograms, common blocks, and formal arguments of the current subprogram. The **PASCAL** attribute cannot be used on formal arguments. This attribute identifies an argument or subprogram as having the characteristics of Microsoft Pascal:

- The argument or the subprogram's arguments are passed by value (unless the **REFERENCE** attribute is specified).
- Microsoft FORTRAN's calling conventions are still used.

2.6.8 REFERENCE

The **REFERENCE** attribute can only be used on formal arguments of the current subprogram. The **REFERENCE** attribute specifies that the argument's location in memory is to be passed, rather than the argument's value. This is called passing by reference.

2.6.9 VALUE

The **VALUE** attribute can only be used on formal arguments of the current subprogram. The **VALUE** attribute specifies that the actual argument's value is to be passed. This is called passing by value. Although assignment to the formal argument is still permitted, values cannot be returned through the argument. Items passed to arguments with the **VALUE** attribute can be of different data types. If type conversion is necessary, it is performed before the call, following the rules discussed in Section 2.7.1.2, "Type Conversion of Arithmetic Operands." If either **C** or **PASCAL** is specified on the subprogram definition, all the arguments are assumed to have the **VALUE** attribute. Character values, substrings, assumed-size arrays, and adjustable-size arrays cannot be passed by value.

In C, arrays are never passed by value. If you specify the **C** attribute and your subprogram has an array argument, the array will be passed as if it were a C data structure (**struct**). To pass an array and have it treated as an array (instead of as a **struct**), you can do one of two things:

1. Use the **REFERENCE** attribute on the formal argument.
2. Pass the result of the **LOC**, **LOCNEAR**, or **LOCFAR** functions by value.

■ Example

```
! In the following example, the integer x is passed by
! value:

      SUBROUTINE Y1X[VALUE]
      INTEGER x[VALUE]
```

2.6.10 VARYING

In FORTRAN, a formal argument must be defined for each actual argument. Some other languages, however, such as C, allow actual arguments for which no formal arguments are defined. These actual arguments are assumed to be passed by value, with no automatic data-type conversions. When the **C** attribute is specified you can also specify the **VARYING** attribute. It means that the number of actual arguments may be different from the number of formal arguments. Actual arguments for which a formal argument is defined must follow the type rules.

When writing a FORTRAN procedure with the **VARYING** attribute, make sure that your code only executes references to arguments you passed in the call, or you will get undefined results.

Note that the FORTRAN Pascal calling sequence cannot support varying numbers of arguments; the **VARYING** attribute has no effect unless you have also specified the **C** attribute on the subprogram.

2.7 Expressions

An expression is a formula for computing a value. Expressions consist of operands and operators. The operands can be function invocations, variables, constants, or other expressions. The operators specify the actions to be performed on the operands. In the following expression, for example, the slash (/) is an operator and `chickens` and `coops` are operands:

```
chickens/coops
```

There are four kinds of expressions in FORTRAN:

1. Arithmetic expressions
2. Character expressions
3. Relational expressions
4. Logical expressions

Each type of expression works with certain types of operands and uses a specific set of operators. Evaluation of an expression produces a value of a specific type.

Expressions are components of statements. In the following example, the entire line is a statement; only the portion after the equal sign (=) is an expression:

```
cost = 10.95*chickens/coops
```

Any variable, array element, or function that is referred to in an expression must be defined at the time the reference is made, or you will receive undefined results. Integer variables must be defined with an arithmetic value, rather than a statement-label value set by an **ASSIGN** statement. FORTRAN only guarantees that expressions generate correct values, not that all parts of expressions are evaluated. For example, if you use an expression that multiplies `(37.8/scale**expo + factor)` by zero, then `(37.8/scale**expo + factor)` may not be evaluated. Similarly, if a false value and another expression are operands of the **.AND.** operator, that expression may not be evaluated. In the following expression, for example, the expression `(SWITCH.EQ.ON)` may not be evaluated:

```
((3.LE.1) .AND. (SWITCH.EQ.ON))
```

2.7.1 Arithmetic Expressions

An arithmetic expression produces a value that is an integer or a real or complex number. The basic operands used in arithmetic expressions are

- Arithmetic constants
- Symbolic names for arithmetic constants
- Variable references
- Array-element references
- Function references

The value of a variable or array element must be defined before it can be referenced in an arithmetic expression or you will receive undefined results. 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 basic operands in the preceding list using parentheses and the arithmetic operators shown in Table 2.5.

Table 2.5

Arithmetic Operators

Operator	Operation	Precedence
**	Exponentiation	1 (highest)
/	Division	2
*	Multiplication	2
–	Subtraction (binary) or negation (unary)	3
+	Addition (binary) or identity (unary)	3

All of the arithmetic operators may be used as binary operators, which appear between two operands. The plus and minus operators can also be used as unary operators, which precede their single operands.

When consecutive operations are of equal precedence, the leftmost operation is performed first. For example, the expression `first/second*third` is equivalent to `(first/second)*third`. There is one exception to this rule: exponentiation. When there are two consecutive exponentiation operations, the rightmost operation is performed first. For example, the following expressions are equivalent:

```
first**second**third
first**(second**third)
```

FORTRAN prohibits two arithmetic operators from appearing consecutively. For example, FORTRAN does not allow `first**-second`, but it does permit `first**(-second)`.

Use parentheses in expressions to control the order in which operators are evaluated. The following list shows some examples of precedence of arithmetic operators:

Expression	Equivalent Expression
`-one**two`	`-(one**two)`
`+x/y`	`+(x/y)`
`area/g-qu**2**factor`	`(area/g)-(qu**(2**factor))`

The following arithmetic operations are prohibited:

- Dividing by 0
- Raising a 0-value operand to a negative power
- Raising a negative-value operand to a nonintegral real power

2.7.1.1 Integer Division

When two integers are divided, the result is the mathematical quotient of the two values, truncated. Thus, `7/3` evaluates to 2, `(-7)/3` evaluates to `-2`, and both `9/10` and `9/(-10)` evaluate to 0.

For example, look at the following assignment statement:

```
X = 1/4 + 1/4 + 1/4 + 1/4
```

First, note that division has higher precedence than addition, so the expression is equivalent to `(1/4) + (1/4) + (1/4) + (1/4)`. Then, take the quotient of `1/4` and truncate toward 0. The result of integer division of 1 by 4 is 0. Therefore, the assignment statement sets `X` equal to 0.

2.7.1.2 Type Conversion 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 exception to the rule is operations involving both **REAL*8** numbers and **COMPLEX*8** numbers, which yield **COMPLEX*16** results.

The ranking of arithmetic operands is as follows:

1. **COMPLEX*16** (highest rank)
2. **COMPLEX*8**
3. **REAL*8** or **DOUBLE PRECISION**
4. **REAL*4**
5. **INTEGER*4**
6. **INTEGER*2**
7. **INTEGER*1** (lowest rank)

For example, when an operation is performed on an **INTEGER*2** operand and a **REAL*4** operand, the **INTEGER*2** is first converted to **REAL*4**. The result of the operation is also a value of data type **REAL*4**. Similarly, if you perform an operation on a real number and a complex number, the real number is first converted to a complex number, and the result of the operation is also complex.

The following list shows some examples of how expressions are interpreted. The variables `i1` and `i2` are integers, `r1` and `r2` are single-precision real numbers, and `c1` and `c2` are **COMPLEX*8** numbers.

Statement	Interpretation
`r2 = i1/i2*r1`	First, integer division (as described in Section 2.7.1.1) is performed on `i1` and `i2`. Then the result is converted to a real number and real multiplication is performed on the resulting operand and `r1`.
`c1=c2+i1`	The integer `i1` is first converted to type **COMPLEX*8**. Then `i1` is added to `c2`.

For expressions with integer operands, note that the type of the result is the maximum of the operand precisions and the setting of the $STORAGE metacommand. For example, if you declare J and K as INTEGER*2 variables, and don't set the $STORAGE metacommand, the result of the expression J + K is an INTEGER*4. Since the default for $STORAGE is 4, the maximum of the operand precisions (both are 2 bytes) and the setting of the $STORAGE metacommand (which defaults to 4 bytes) is 4 bytes.

For information on passing integers as arguments, see Section 3.6, "Arguments."

Note also that the compiler usually removes higher-precision arithmetic when optimizing, if it will not affect the result, and if $DEBUG is not set. For example, if $DEBUG is not set, the arithmetic in the following example is probably 16-bit arithmetic, even if $STORAGE:4 was specified:

```
INTEGER*2 I,J,K
I=J+K
```

Using $STORAGE:4 does not affect INTEGER*2 expressions which have only the plus (+), minus (-), or multiplication (*) operators in them. The results are the same in either 16- or 32-bit arithmetic, since any overflows that occur are ignored if $DEBUG is not set. Table 2.6 shows how arithmetic operands are converted from one data type to another. For example, to see how a REAL*8 number is converted to a REAL*4 number, first find REAL*8 in the "Data Type" column. Now look under "Converting to the Next-Lower-Ranked Data Type." REAL*8 numbers are converted to REAL*4 numbers by rounding off the least-significant part.

Table 2.6

Arithmetic Type Conversion

Data Type	Converting to the Next-Higher-Ranked Data Type	Converting to the Next-Lower-Ranked Data Type
COMPLEX*16 (highest rank)	--------	Convert imaginary and real parts, individually, from REAL*8 to REAL*4.[1]
COMPLEX*8	Convert imaginary and real parts, individually, from REAL*4 to REAL*8.	Delete imaginary part.
DOUBLE PRECISION	Convert from **DOUBLE PRECISION** to **REAL*4** and add a 0.0 imaginary part.[1]	Round off least-significant part.
REAL*4	Store in the **DOUBLE PRECISION** format.	Truncate.
INTEGER*4	Add zero fractional part.	Use least-significant part.
INTEGER*2	Use as least-significant part, and set sign bit in most-significant part.	Use least-significant part.
INTEGER*1 (lowest rank)	Use as least-significant part, and set sign bit in most-significant part.	--------

[1] REAL*8 numbers are converted to COMPLEX*16 by adding a 0.0 imaginary part, and COMPLEX*16 numbers are converted to REAL*8 by deleting the imaginary part. They are not first converted to COMPLEX*8 numbers.

2.7.2 Character Expressions

A character expression produces a value that is of type character. There are five basic operands used in character expressions:

1. Character constants
2. Character variable references

3. Character array-element references
4. Character function references
5. Character substrings

There is only one character operator: the concatenation operator (**//**). The operator is used as follows:

first **//** *second*

This produces a character string whose value is the value of *first* concatenated on the right with the value of *second* and whose length is the sum of the lengths of *first* and *second*. For example, the following expression produces the string `'ABCDE'`:

```
'AB' // 'CDE'
```

If you concatenate C strings, remember that a null character (`\0`) is automatically appended to the end of each C string. For example, look at the following expression:

```
'hello'C // 'world'C
```

It is equivalent to the following C string:

```
'hello\0world'C
```

2.7.3 Relational Expressions

Relational expressions compare the values of two arithmetic or character expressions. If an arithmetic expression is compared with a character expression, the arithmetic expression is considered to be a character expression. If the **SSTRICT** metacommand is specified, you cannot compare an arithmetic variable with a character variable. Comparison of arithmetic and character variables is prohibited by the ANSI standard.

The result of a relational expression is of type **LOGICAL**. Relational expressions can use any of the operators shown in Table 2.7 to compare values.

Table 2.7

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. This is because a relational expression cannot contain another relational expression. For example, look at the following program fragment:

```
C THIS IS INCORRECT
      REAL*4 A,B,C
      IF((A .LT. B) .NE. C) D=12
```

Assume that A is less than B. After the first part of the expression is evaluated, the expression is as follows:

```
.TRUE. .NE. C
```

But C is an arithmetic expression, and you cannot compare an arithmetic expression to **.TRUE.**. To compare relational expressions and logical values, use the logical operators (discussed in Section 2.7.4).

Relational expressions with arithmetic operands may have one operand that is an integer and one that is a real number. In this case, the integer operand is converted to a real number before the relational expression is evaluated. You can also have a complex operand, in which case the other operand is first converted to complex. However, you can use only the **.NE.** and **.EQ.** operators with complex operands.

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. For example, `('ABCDEFG'.LT.'BCDEFGH')` returns the value **.TRUE.**, and

('Keith' .GE. 'Susan') returns the value **.FALSE.**. If operands of unequal length are compared, the shorter operand is extended to the length of the longer operand by the addition of blanks on the right.

2.7.4 Logical Expressions

A logical expression produces a logical value. There are five basic operands used in logical expressions:

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 basic operands in the preceding list by using parentheses and the logical operators of Table 2.8.

Table 2.8

Logical Operators

Operator	Operation	Precedence
.NOT.	Negation	1 (highest)
.AND.	Conjunction	2
.OR.	Inclusive disjunction	3
.EQV.	Equivalence	4
.NEQV.	Nonequivalence	4

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. If switch, for example, is **.TRUE.**, then (.NOT. switch) is **.FALSE.**.

When two consecutive operations are of equal precedence, the leftmost operation is performed first.

Two **.NOT.** operators cannot be adjacent to each other, but the **.NOT.** operator can appear next to any of the other logical operators. The following statement, for example, is allowed:

```
logvar = a .AND. .NOT. b
```

Logical operators have the same meaning as in standard mathematical semantics; the **.OR.** operator is nonexclusive. For example,

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

evaluates to the value **.TRUE.**. Table 2.9 shows the values of logical expressions.

Table 2.9

Values of Logical Expressions

If Operands a and b Are the Following:	Then These Expressions Evaluate as Follows:			
	a .AND. b	**a .OR. b**	**a .EQV. b**	**a .NEQV. b**
Both true	True	True	True	False
One true and one false	False	True	False	True
Both false	False	False	True	False

■ Example

The following program fragment demonstrates precedence in logical expressions:

```
      LOGICAL stop, go, wait, a, b, c, d, e
C The following two statements are equivalent:
      stop =  a .AND. b  .AND. c
      stop = (a .AND. b) .AND. c
C The following two statements are equivalent:
      go =   .NOT. a  .OR. b  .AND. c
      go = ((.NOT. a) .OR. b) .AND. c
C The following two statements are equivalent:
      wait =   .NOT. a  .EQV. b .OR. c   .NEQV. d .AND. e
      wait = ((.NOT. a) .EQV. (b .OR. c)) .NEQV. (d .AND. e)
```

2.7.5 Precedence of Operators

When arithmetic, character, relational, and logical operators appear in the same expression, precedence is as follows:

1. Arithmetic operators have the highest precedence.
2. Character operators are evaluated next.
3. Relational operators are evaluated next.
4. Logical operators have the lowest precedence.

Chapter 3
Program Structure

3.1 Introduction

This chapter explains how to structure your FORTRAN programs. First it discusses the format of programs. Then it explains arguments and describes the types of program units available in FORTRAN.

3.2 Lines

Lines in a FORTRAN program are also called source records or cards. Lines are composed of sequences of characters. Characters are interpreted differently depending on what column they are in, as shown in the following list:

Column	Character Interpretation
1 – 5	Statement label. A dollar sign ($) in column 1 indicates a metacommand. An asterisk or an uppercase or lowercase C in column 1 indicates a comment line.
6	Continuation character.
7 – 72	FORTRAN statement.
73 and above	Ignored.

Lines shorter than 72 characters are padded with blanks.

There are five kinds of lines in Microsoft FORTRAN:

Type of Line	Description
Comment lines	A comment line has an uppercase or lowercase C or an asterisk (*) in column 1, or the line is entirely blank. Comment lines do not affect the execution of the FORTRAN program in any way. Comment lines can appear within statements that have continuation lines. Note that debug lines, described below, are sometimes treated as comment lines. For more information, see Section 6.2.1, "The $DEBUG and $NODEBUG Metacommands." The following are examples of comment lines:

```
C This is a comment line,
*                 and so is this.
```

Initial lines

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

Except for the statement following a logical **IF**, all FORTRAN statements begin with an initial line. The following are initial lines:

```
      GOTO 100
00002 CHARACTER*10 name
100   0CONTINUE
  1000STOP ' '
```

Metacommand

A metacommand line has a dollar sign ($) in column 1.

Metacommands control the operation of the Microsoft FORTRAN Compiler. The following lines are metacommands:

```
$DEBUG
$LINESIZE:132
```

See Chapter 6 for more information on metacommands.

Continuation lines

A continuation line is any line having blanks in columns 1 through 5 and a character (other than a blank or a zero) in column 6. A continuation line increases the amount of room in which to write a statement. A statement may be extended to include as many continuation lines as memory allows. If the **$STRICT** metacommand is set, a warning is generated if more than 19 consecutive continuation lines are used. The second line below is a continuation line:

```
      INTEGER*4 count, popu, local,
     +ovrflo, incrs, provnc
```

Debug lines

A debug line contains any uppercase or lowercase alphabetic character, except C, c, and the dollar sign, in column 1. If the **$DEBUG** metacommand specifies the character that is

in column 1 of a debug line, that character is treated as a blank character and the line is compiled like any other line. If the **$DEBUG** metacommand does not specify the character that is in column 1, that line is treated like a comment line. See Section 6.2.1 for more information on debug lines. The following lines are debug lines:

```
D       DEBUG line
d       xDEBUG line
```

3.3 Statement Labels

Any statement can start with a label; however, only the labels of executable or **FORMAT** statements can be referenced. 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, and blanks and leading zeros are not significant. Each labeled statement in a source file must have a unique label.

3.4 Free-Form Source Code

By specifying the **$FREEFORM** metacommand, you can write your source code in the free-form format. Most of the rules specified in Section 3.2, "Lines," do not apply to the free-form format. The following rules define the free-form format:

- A double quotation mark (") in column 1 indicates a comment line.
- Initial lines may start in any column.
- The first nonblank character of an initial line may be a digit; the first digit in a statement label. The statement label may be from one to five decimal digits; blanks and leading zeros are ignored. Blanks are not required to separate the statement label from the first character of the statement.
- If the last nonblank character of a line is a minus sign, it is discarded and the next line is taken to be a continuation line. The continuation line may start in any column.

- Alphabetic characters and asterisks are not allowed as comment markers in column 1.

See Section 6.2.5, "The $FREEFORM and $NOFREEFORM Metacommands," and the *Microsoft FORTRAN Compiler User's Guide* for more information on free-form format.

3.5 Order of Statements and Metacommands

Statements describe, specify, and classify the elements of your program, as well as the actions your program will take. Chapter 5, "Statements," defines each Microsoft FORTRAN statement.

The ANSI standard for the FORTRAN language enforces a certain ordering of the statements and lines that make up a FORTRAN program unit. Some of Microsoft FORTRAN's extensions have additional requirements. Figure 3.1 shows which statements and metacommands must precede, and which must follow, any specific statement or metacommand.

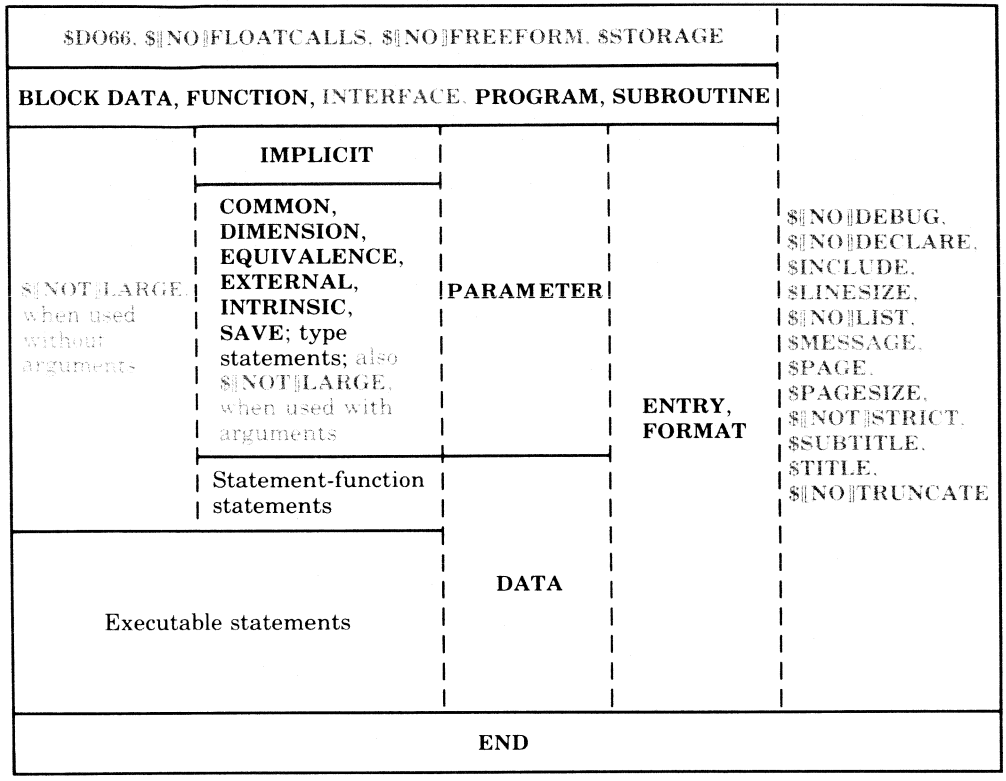

Figure 3.1 Order of Statements and Metacommands

Suppose you have a program that contains program elements *a* and *b*. In Figure 3.1, if the box containing program element *a* is above the box containing program element *b*, then *a* must appear before *b* in your program. **IMPLICIT**, for example, is above **COMMON**, **DATA**, **END**, and so on. So, if you have an **IMPLICIT** statement in your program, it must appear before any of those statements.

If, in Figure 3.1, *a* is in a box that is to the left or right of *b*, then *a* and *b* can appear in any order relative to each other. **FORMAT**, for example, is to the left of the box containing most of the metacommands, and is to the right of the box containing **PARAMETER**, **IMPLICIT**, **COMMON**, statement-function statements, **DATA**, and so on. Thus, in your program, any of those elements can appear before or after a **FORMAT** statement.

The following rules summarize the required order of statements and metacommands shown in Figure 3.1:

- Every program unit must have an **END** statement as its last line.
- Comment lines can appear anywhere, except after the last **END** statement in a source file.
- **BLOCK DATA**, **FUNCTION**, INTERFACE, **PROGRAM**, and **SUBROUTINE** statements must precede all other statements. They do not have to precede metacommands.
- All specification statements must precede all **DATA** statements, statement-function statements, and executable statements. See Section 5.2, "Categories of Statements," for listings of specification statements and executable statements.
- **IMPLICIT** statements must precede other specification statements, with the exception of the **PARAMETER** statement.
- Statement-function statements must precede executable statements.
- When a specification statement defines the type of a constant to be used in the **PARAMETER** statement, the **PARAMETER** statement must follow that specification statement. The **PARAMETER** statement must precede all other specification statements that use the symbolic constants it defines.
- INTERFACE statements must precede references to the subprograms they define.
- The $D066, $[NO]FLOATCALLS, $[NO]FREEFORM and $STORAGE metacommands, if present, must appear before anything else. $LARGE and $NOTLARGE, when used without arguments, cannot appear within the executable statement section. $LARGE and $NOTLARGE, when used with arguments, must appear in the declarative section. Other metacommands can appear anywhere.
- Block-data subprograms may not contain statement-function statements, **FORMAT** statements, or executable statements.

3.6 Arguments

Arguments are values passed to and from functions and subroutines. A formal argument is the name by which an argument is known within a function or subroutine. An actual argument is the specific variable, expression, array, or other item passed to a subroutine or function at any specific calling location.

By default, arguments pass values into and out of subroutines or functions by reference; that is, they pass the memory address of the argument. You can use the VALUE attribute (described in Section 2.6.9) to pass arguments by value. The number of actual arguments must be the same as the number of formal arguments (unless the VARYING attribute, described in Section 2.6.10, is specified), and the corresponding types must agree. Defining a subroutine or function prior to its first use, or using the INTERFACE statement (described in Section 5.3.34), causes the compiler to check for the correct number and type of arguments. Otherwise, the compiler uses the arguments that are given the first time the subroutine or procedure is executed to determine the number and type of arguments. Note that the compiler can detect incorrect matching between actual and formal argument types only if the type of the formal argument is known.

Upon entry to a subroutine or function, the actual arguments are associated with the formal arguments. This association remains in effect until execution of the subroutine or function is terminated. If the actual argument has been passed by reference (the default), assigning a value to a formal argument during execution of a subroutine or function can alter the value of the corresponding actual argument.

If an actual argument is a constant, a function reference, or an expression other than a single variable, assigning a value to the corresponding formal argument is not permitted and has unpredictable results. In the following program, for example, the actual argument `header` is a constant and corresponds to the formal argument `title`. In the subroutine `report`, a value is assigned to `title`:

```
C This program is incorrect and has unpredictable
C results.
      character*20 header
      real*4 grav
C header is a constant
      parameter (header = 'Specific Gravity')
      data grav /2.8327/
      write (*,*) header, grav
C header is an actual argument
      call report (header, grav)
      write (*,*) header, grav
      stop ' '
      end
C The formal argument corresponding to header is title
      subroutine report (title, data)
      character*20 title
      real*4 data
C The following statement is illegal because it assigns
C a value to a formal argument that corresponds to
C a constant
      title = 'Density (kg/cubic m)'
      write (*,*) title, data
      return
      end
```

The output of the above program is unpredictable. To change the value of `title` in the subroutine, `header` could have been made a variable, instead of a constant.

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. The subscript expressions remain constant throughout the execution of the subroutine or function, even if they contain variables that are redefined during the execution of the subroutine or function.

The following list indicates how you can associate actual and formal arguments:

Actual Argument	Formal Argument
A variable, an array element, or an expression.	Variable name
An alternate-return specifier (`*`*n*) in the **CALL** statement. See Section 5.3.5 for an explanation of alternate-return specifiers. Note that a formal argument that is an alternate return is repeatable.	An asterisk (`*`)

Array name	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. Note that a reference to an element outside these bounds is not detected as an error, and the results of an out-of-bounds subscript 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 subroutine or function 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 subroutine argument or function argument. The intrinsic function must have been declared with an **INTRINSIC** statement in the program unit where it is used as an actual argument. The following intrinsic functions may *not* be associated with formal subroutine arguments or function arguments:

AMAX0	**DMIN1**	**INT**	**LLT**	**MIN**
AMAX1	DREAL	INT1	LOC	**MIN0**
AMIN0	EOF	INT2	LOCFAR	**MIN1**
AMIN1	**FLOAT**	INT4	LOCNEAR	**REAL**
CHAR	HFIX	INTC	**LOG**	**SNGL**
CMPLX	**ICHAR**	JFIX	**LOG10**	
DBLE	**IDINT**	**LGE**	**MAX**	
DCMPLX	**IFIX**	**LGT**	**MAX0**	
DMAX1	IMAG	**LLE**	**MAX1**	

When passing integer arguments, note that an **INTEGER*2** variable cannot be passed to a formal **INTEGER*4** argument, and an **INTEGER*4** variable cannot be passed to a formal **INTEGER*2** argument. You can convert the data type using the intrinsic functions **INT4** or **INT2** (described in Section 3.11.3.1), but the result is an expression and the subroutine cannot assign anything to the argument. Also, note that when **$STORAGE:4** (the default) is in effect, even expressions with only **INTEGER*2** arguments have a result with type **INTEGER*4**. The following program, for example, results in an error:

If the definition of a subprogram precedes a **CALL** statement or function reference in the same source file that references that subprogram, the compiler checks 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.

3.7 Program Units

The Microsoft FORTRAN Compiler processes program units. A program unit can be a main program, a subroutine, a function, or a block-data subprogram. You can compile any of these units separately and link them together later. It is not necessary to compile or recompile them as a whole. The following list summarizes the four types of program units (discussed in Sections 3.8 through 3.11):

Program Unit	Description
Main program	Any program unit that does not have a **FUNCTION**, **SUBROUTINE**, or **BLOCK DATA** statement as its first statement. A main program can have a **PROGRAM** statement as its first statement, but this is not required.
Subroutine	A program unit that can be called from other program units by a **CALL** statement.
Block-data subprogram	A program unit that provides initial values for variables in named common blocks.
Function	A program unit that can be referred to in an expression.

External functions, subroutines, and block-data subprograms are collectively called subprograms. Subroutines and functions are collectively called procedures. The **PROGRAM**, **SUBROUTINE**, **BLOCK DATA**, **FUNCTION**, and statement-function statements are described in detail in Section 5.3, "Statement Directory." Related information is provided in the entries for the **CALL** and **RETURN** statements.

By using subprograms you can develop large, more structured programs. This is useful in the following situations:

If:	Then:
You have a large program	You can more easily develop, test, maintain, and compile a large program when it is broken into parts.
You intend to include certain routines in more than one program	You can create object files that contain these routines and link them to each of the programs in which the routines are used.
You would like a routine to be implemented in several different ways	You can place the routine in its own file and compile it separately. Then, to improve performance, you can alter the implementation or even rewrite the routine in assembly language, Microsoft Pascal, or Microsoft C. The rest of your program will not need to change.

3.8 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. Main programs are always assigned the global name _main. The name _main should not be used for anything else in a program. The name "main" is actually permitted as a common-block name, as a local variable in a subprogram outside the main program, or as a subprogram name in a module that does not contain a PROGRAM statement and is not referenced by a module that contains the main program. However, because it is likely to cause confusion, the use of "main" is not recommended. If the main program has a PROGRAM statement, the name specified in the PROGRAM statement is assigned in addition to the name _main.

The execution of a program always begins with the first executable statement in the main program, so there must be exactly one main program in every executable program.

For further information about programs, see Section 5.3.42, "The PROGRAM Statement."

3.9 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. Then the subroutine returns control to the statement immediately following the one that called it or to a statement specified as an alternate return. See Section 5.3.5, "The CALL Statement," for more information.

A subroutine does not directly return a value. However, values can be passed back to the calling program unit through arguments or common variables.

See Section 5.3.50, "The SUBROUTINE Statement," for further information.

3.10 Block-Data Subprograms

A block-data subprogram is a program unit that provides initial values for variables in common blocks.

Variables are normally initialized with **DATA** statements. Variables in named common blocks can be initialized only in block-data subprograms. Variables in blank common blocks cannot be initialized in block-data subprograms. See Section 5.3.4, "The BLOCK DATA Statement," for more information.

3.11 Functions

A function is referred to in an expression and returns a value that is used in the computation of that expression. Functions can also return values through arguments and common variables. There are three kinds of functions:

1. External functions
2. Statement functions
3. Intrinsic functions

Each of these is described in more detail in Sections 3.11.1 – 3.11.3.

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 is used as an operand in the expression containing the function reference. Note that if a character function is referenced in a program unit, the function length specified in the program unit must be an integer-constant expression.

The syntax of a function reference is as follows:

fname ([*arguments*])

Parameter	Value
fname	Name of an external, intrinsic, or statement function.
arguments	Actual arguments. If more than one argument is given, they must be separated by commas.

The rules for arguments for functions are identical to those for subroutines (except that alternate returns are not allowed), and are described in Section 5.3.5, "The CALL Statement." Some additional restrictions specific to statement functions and intrinsic functions are described in Section 3.11.2, "Statement Functions," and Section 3.11.3, "Intrinsic Functions."

3.11.1 External Functions

An external function is specified by a function program unit. It begins with a **FUNCTION** statement and concludes with an **END** statement.

■ Example

```
C The following example is an external function that
C calculates the Simpson approximation of a definite
C integral
      INTEGER FUNCTION simpson(delx,steps,y)
      INTEGER delx,y(100),sum,steps,factor
      sum=0
      DO 100 i=0,steps
        IF((i.EQ.0).OR.(i.EQ.steps))THEN
          factor=1
        ELSEIF(MOD(i,2).EQ.0)THEN
          factor=2
        ELSE
          factor=4
        ENDIF
        sum=factor*y(i)+sum
100   CONTINUE
      simpson=INT((REAL(delx)/3.)*REAL(sum))
      END
```

3.11.2 Statement Functions

A statement function is defined by a single statement and is similar in form to an assignment statement.

A statement function is not an executable statement, because it is not executed in order as the first statement in its particular program unit. The body of a statement function defines the meaning of the statement function. It is carried out, like other functions, by executing a function reference in an expression.

For information on the syntax and use of a statement-function statement, see Section 5.3.48, "The Statement-Function Statement."

3.11.3 Intrinsic Functions

Intrinsic functions are predefined by the Microsoft FORTRAN language. Appendix B summarizes all the intrinsic functions available in Microsoft FORTRAN. Other functions supplied with Microsoft FORTRAN are described in Appendix C, "Additional Procedures."

An **IMPLICIT** statement cannot change the type of an intrinsic function. For those generic intrinsic functions that allow several types of arguments, all arguments in a single reference should be of the same type. The intrinsic function **IDIM**, for example, takes two integer arguments and returns the positive difference. If you specify `IDIM(I,J)`, `I` and `J` should both be of the same type. If, however, two arguments are of different data types, the Microsoft FORTRAN Compiler first attempts to convert the arguments to the correct data type. For instance, if `I` and `J` in the example above are real numbers, they are first converted to the **INTEGER** type (see Table 2.6, "Arithmetic Type Conversion," for information on type conversion), and then the operation takes place. Or, if `I` is of type **INTEGER*2** and `J` is of type **INTEGER*4**, `I` is first converted to **INTEGER*4**, and then the operation takes place.

Intrinsic-function names, except those listed in Section 3.6, "Arguments," can appear in an **INTRINSIC** statement. Intrinsic functions specified in **INTRINSIC** statements can be used as actual arguments in external procedure references. An intrinsic-function name can also appear in a type statement, but only if the type is the same as the standard type for that intrinsic function.

Arguments must agree in order, number, and type with those specified in Tables 3.2 – 3.17. Arguments can also be expressions of the specified type.

"Generic" intrinsic functions, like **ABS**, allow you to use the same intrinsic-function name with more than one type of argument. "Specific" intrinsic functions, like **IFIX**, can only be used with one type of argument. For example, specific intrinsic functions can be useful if you want to ensure that 16-bit arithmetic is performed in an expression, or if you want to guarantee the range of your result.

If an intrinsic function's name is in the formal argument list of a function or subroutine in a subprogram, then that name is being used for something else in that subprogram; it is, therefore, not an intrinsic function. In the following statement, for example, `sign`, `float`, and `index` are not being used as intrinsic functions:

```
SUBROUTINE process (sign, float, index)
```

If you give an argument for which the result is not mathematically defined, or for which the result exceeds the numeric range of the processor, the result of the intrinsic function is undefined.

The result of an intrinsic function of complex type is the principal value. The principal value of a complex number is the number whose argument is less than or equal to π and greater than $-\pi$.

When results of generic integer intrinsic functions are passed to subprograms, the setting of **SSTORAGE** determines the data type of the value to be passed.

Table 3.1 explains the abbreviations used for all 15 tables of intrinsic functions in this chapter.

Table 3.1

Abbreviations Used to Describe Intrinsic Functions

Abbreviation	Data Type
gen	More than one possible argument type; see "Argument Type" column
int	**INTEGER**, INTEGER*1, INTEGER*2, or INTEGER*4
int1	INTEGER*1
int2	INTEGER*2
int4	**INTEGER***4
real	**REAL**, REAL*4, **DOUBLE PRECISION**, or REAL*8
real4	**REAL***4
dbl	**REAL*8**
log	**LOGICAL**, LOGICAL*1, LOGICAL*2, or LOGICAL*4
log1	LOGICAL*1
log2	LOGICAL*2
log4	**LOGICAL***4
cmp	**COMPLEX**, COMPLEX*8, or COMPLEX*16
cmp8	**COMPLEX***8
cmp16	COMPLEX*16
char	**CHARACTER** [*n]

3.11.3.1 Data-Type Conversion

This section describes how the type-conversion intrinsic functions work. Table 3.2 summarizes the intrinsic functions that perform type conversion.

Table 3.2
Intrinsic Functions: Type Conversion

Name	Argument Type	Function Type
INT(*gen*)	int, real, or cmp	int
INT1(*gen*)	int, real, or cmp	**INTEGER*1**
INT2(*gen*)	int, real, or cmp	**INTEGER*2**
INT4(*gen*)	int, real, or cmp	**INTEGER*4**
INTC(*gen*)	int, real, or cmp	**INTEGER[C]**
IFIX(*real4*)	**REAL*4**	int
HFIX(*gen*)	int, real, or cmp	**INTEGER*2**
JFIX(*gen*)	int, real, or cmp	**INTEGER*4**
IDINT(*dbl*)	**REAL*8**	int
REAL(*gen*)	int, real, or cmp	**REAL*4**
DREAL(*cmplx*)	**COMPLEX*16**	**REAL*8**
FLOAT(*int*)	int	**REAL*4**
SNGL(*dbl*)	**REAL*8**	**REAL*4**
DBLE(*gen*)	int, real, or cmp	**DOUBLE PRECISION**
CMPLX(*genA*[,*genB*])	int, real, or cmp	**COMPLEX*8**
DCMPLX(*genA*[,*genB*])	int, real, or cmp	**COMPLEX*16**
ICHAR(*char*)	char	int
CHAR(*int*)	int	char

Note: See Table 3.1 for a list of the abbreviations used in this table.

The **INT** intrinsic function converts arguments to integers. If the argument *gen* is an integer, then **INT**(*gen*) equals *gen*. If *gen* is real, then **INT**(*gen*) equals the value of *gen*, truncated. `INT(1.9)`, for example, is equal to 1, and `INT(-1.9)` is equal to −1. If *gen* is complex, first the real part of *gen* is taken; then, that real part is converted to an integer by truncation. **INT1** converts its arguments to **INTEGER*1**. **INT2** and **HFIX** convert their arguments to **INTEGER*2**. **INT4** and **JFIX** convert their arguments to **INTEGER*4**. They can be used to convert the data type of an expression or variable to an expression of the correct type for passing as a subprogram argument. **INT2** can also be used to direct the compiler to use short arithmetic in expressions which would otherwise be long, and **INT4** can specify long arithmetic in expressions which would otherwise be short.

The **INTC** intrinsic function converts arguments to C integers. C integers are described in Section 2.6.2, "C." The **IFIX** and **IDINT** intrinsic functions convert single- or double-precision arguments, respectively, to integers.

The **REAL** intrinsic function converts numbers to the single-precision real data type. If *gen* is an integer, then **REAL**(*gen*) is *gen* stored as a single-precision real number. If *gen* is a single-precision real number, then **REAL**(*gen*) equals *gen*. If *gen* is complex, then **REAL**(*gen*) equals the real part of *gen*. If *gen* is a double-precision number, then **REAL**(*gen*) is the first six significant digits of *gen*.

The **DBLE** intrinsic function converts numbers to the double-precision real data type. The **FLOAT** and **SNGL** intrinsic functions convert numbers to the single-precision real data type. They work like the **REAL** intrinsic function. The DREAL intrinsic function converts **COMPLEX*16** numbers to the double-precision real data type by deleting the imaginary part.

The **CMPLX** and DCMPLX intrinsic functions convert numbers to the complex data types. If only one argument, *gen*, is specified, *gen* can be an integer, real, double-precision, or complex number. In this case, if *gen* is complex, **CMPLX**(*gen*) equals *gen*. If *gen* is an integer, real, or double-precision number, the real part of the result equals **REAL**(*gen*), and the imaginary part equals 0.0. If two arguments are specified, *genA* and *genB* must both be the same type, and they can be integers, real numbers, or double-precision numbers. In this case, the real part of the result equals **REAL**(*genA*), and the imaginary part of the result equals **REAL**(*genB*).

The **ICHAR** intrinsic function translates characters into integers, and the **CHAR** intrinsic function translates integers into characters. The ASCII character sequence is used to make the translation, as shown in Appendix A, "ASCII Character Codes." Both the argument of the **CHAR** intrinsic function and the result of the **ICHAR** intrinsic function must be greater than or equal to 0, and less than or equal to 255. The argument of the **ICHAR** intrinsic function must be a single character.

Note

For two characters, `charA` and `charB`, the expression `(charA.LE.charB)` is true if and only if `(ICHAR(charA).LE.ICHAR(charB))` is true.

■ Example

The following list shows examples of the type-conversion intrinsic functions:

Function Reference	Equivalent
`INT(-3.7)`	`-3`
`INT(7.682)`	`7`
`INT(0)`	`0`
`INT((7.2,39.3))`	`7`

3.11.3.2 Truncating and Rounding

Table 3.3 summarizes the intrinsic functions that perform truncation and rounding.

Table 3.3
Intrinsic Functions: Truncation and Rounding

Name	Truncate or Round	Argument Type	Function Type
AINT(*real*)	Truncate	real	Same as argument
DINT(*dbl*)	Truncate	**REAL*8**	**REAL*8**
ANINT(*real*)	Round	real	Same as argument
DNINT(*dbl*)	Round	**REAL*8**	**REAL*8**
NINT(*real*)	Round	real	int
IDNINT(*dbl*)	Round	**REAL*8**	int

Note: See Table 3.1 for a list of the abbreviations used in this table.

The intrinsic functions **AINT** and **DINT** truncate their arguments. The intrinsic functions **ANINT**, **DNINT**, **NINT**, and **IDNINT** are evaluated as follows:

Argument	Result
Greater than zero	**INT**(*gen* **+ 0.5**)
Equal to zero	Zero
Less than zero	**INT**(*gen* **− 0.5**)

■ Examples

The following list shows examples of truncation and rounding:

Function Reference	Equivalent
AINT(2.5)	2.0
AINT(-2.5)	-2.0
ANINT(2.5)	3.0
ANINT(-2.5)	-3.0

The following program uses the **ANINT** intrinsic function to perform rounding:

```
C This program adds tax to a purchase amount
C and prints the total.

      REAL AMOUNT, TAXRATE, TAX, TOTAL
      TAXRATE = 0.079
      AMOUNT = 12.99
C Calculate tax and round to nearest hundredth.
      TAX = ANINT(AMOUNT*TAXRATE * 100.)/100.
      TOTAL = AMOUNT + TAX
      WRITE (*,100) AMOUNT, TAX, TOTAL
 100  FORMAT (1X,'AMOUNT ',F7.2/
     1          1X,'TAX    ',F7.2/
     2          1X,'TOTAL  ',F7.2)
      STOP ' '
      END
```

3.11.3.3 Absolute Value and Sign Transfer

Table 3.4 summarizes the intrinsic functions that take absolute values and perform sign transfers.

Table 3.4
Intrinsic Functions: Absolute Values and Sign Transfer

Name	Definition	Argument Type	Function Type
ABS(*gen*)	Absolute value	int, real, or cmp	Function type same as argument type, except when argument is cmp.[1]
IABS(*int*)	Absolute value	int	int
DABS(*dbl*)	Absolute value	**REAL*8**	**REAL*8**
CABS(*cmp*)	Absolute value	cmp	real[1]
CDABS(*cmp16*)	Absolute value	COMPLEX*16	REAL*8
SIGN(*genA*,*genB*)	Sign transfer	int or real	Same as argument
ISIGN(*intA*,*intB*)	Sign transfer	int	int
DSIGN(*dblA*,*dblB*)	Sign transfer	**REAL*8**	**REAL*8**

Note: See Table 3.1 for a list of the abbreviations used in this table.

[1] If argument is **COMPLEX*8**, function is **REAL*4**. If argument is COMPLEX*16, function is REAL*8.

The intrinsic functions **ABS**, **IABS**, **DABS**, **CABS**, and CDABS return the absolute value of their arguments. Note that for a complex number (*real*,*imag*), the absolute value equals the following:

(*real***2 + *imag***2)**(1/2)

For two arguments (genA,genB), the intrinsic functions **SIGN**, **ISIGN**, and **DSIGN** return |genA| if genB is greater than or equal to zero, and −|genA| if genB is less than zero.

■ Examples

The following program uses a sign-transfer intrinsic function:

```
      A = 5.2
      B = -3.1

C The following statement transfers the sign of B to A
C and assigns the result to C.
      C = SIGN(A,B)

C The output will be -5.2.
      WRITE (*,*) C
      STOP ' '
      END
```

The following program uses sign-transfer and absolute-value intrinsic functions:

```
C This program takes the square root of a vector magnitude.
C Since the sign in a vector represents direction, the square
C root of a negative value is not meant to produce complex
C results. This routine removes the sign, takes the square
C root, and then restores the sign.

      REAL      MAG,SGN

      WRITE (*,'(A)') ' ENTER A MAGNITUDE:  '
      READ (*,'(F10.5)') MAG

C Store the sign of MAG by transferring its sign to 1
C and storing the result in SGN.
      SGN = SIGN(1., MAG)

C Calculate the square root of the absolute value of the
C magnitude.
      RESULT = SQRT(ABS(MAG))

C Restore the sign by multiplying the result by -1 or +1.
      RESULT = RESULT * SGN

      WRITE (*,*) RESULT
      STOP ' '
      END
```

The intrinsic functions **DIM**, **IDIM**, and **DDIM** return the positive difference of two arguments as follows:

If:	Then:
genA <= *genB*	**DIM**(*genA*,*genB*) = 0
genA > *genB*	**DIM**(*genA*,*genB*) = *genA* − *genB*

- **Examples**

The following list shows examples of positive differences:

Function Reference	Equivalent
`DIM(10,5)`	`5`
`DIM(5,10)`	`0`
`DIM(10,-5)`	`15`

3.11.3.6 Maximums and Minimums

Table 3.7 summarizes the functions that return the maximum or minimum of two or more values.

3.11.3.4 Remainders

Table 3.5 summarizes the intrinsic functions that return remainde

Table 3.5
Intrinsic Functions: Remainders

Name	Argument Type	Function Type
MOD (*genA*,*genB*)	int or real	Same as argument
AMOD(*real4A*,*real4B*)	**REAL***4	**REAL***4
DMOD(*dblA*,*dblB*)	**REAL*8**	**REAL*8**

Note: See Table 3.1 for a list of the abbreviations used in this table.

The intrinsic functions **MOD**, **AMOD**, and **DMOD** return remainders, as follows:

MOD(*genA*,*genB*) = *genA* − (**INT**(*genA* / *genB*) * *genB*)

If *genB* is 0, the result is undefined.

3.11.3.5 Positive Differences

Table 3.6 summarizes the intrinsic functions that return the positive difference between two arguments.

Table 3.6
Intrinsic Functions: Positive Difference

Name	Argument Type	Function Type
DIM(*genA*,*genB*)	int or real	Same as argument
IDIM(*intA*,*intB*)	int	int
DDIM(*dblA*,*dblB*)	**REAL*8**	**REAL*8**

Note: See Table 3.1 for a list of the abbreviations used in this table.

The following program uses the **MIN** and **MAX** intrinsic functions:

```
C This program uses the MAX intrinsic function to find the
C maximum and minimum elements in a vector x.

        integer    i
        real       x(10), small, large

        data x/12.5,2.7,-6.2,14.1,-9.1,17.5,2.,-6.3,2.5,-12.2/

C Initialize small and large with arbitrarily large and small
C values.
        small = 1e20
        large = -1e2

        do 100, i = 1,10
           small = min(small, x(i))
           large = max(large, x(i))
 100    continue

        write(*,200) small, large
 200    format(' The smallest number was ',f6.1/
     1          ' The largest number was  ',f6.1)
        stop ' '
        end
```

The program above has the following output:

```
The smallest number was  -12.2
The largest number was    17.5
```

3.11.3.7 Double-Precision Products

Table 3.8 lists the intrinsic function that returns a double-precision product.

Table 3.8

Intrinsic Functions: Double-Precision Product

Name	Argument Type	Function Type
DPROD(*real4A*,*real4B*)	**REAL*4**	**REAL*8**

Note: See Table 3.1 for a list of the abbreviations used in this table.

The intrinsic function **DPROD** returns the double-precision product of two single-precision real arguments.

■ Example

The following program uses the **DPROD** intrinsic function:

```
REAL     A,B
REAL*8   C

A = 3.72382
B = 2.39265

WRITE(*,*) A*B,DPROD(A,B)
STOP ' '
END
```

The program above has the following output:

```
 8.9097980          8.90979744044290
```

3.11.3.8 Complex Functions

Table 3.9 lists the intrinsic operations that perform various operations for complex numbers.

Table 3.9

Intrinsic Functions: Complex Operators

Name	Definition	Argument Type	Function Type
AIMAG(*cmp8*)	Imaginary part of **COMPLEX*8** number	**COMPLEX*8**	**REAL*4**
IMAG(*cmp*)	Imaginary part of cmp number	cmp	If argument is COMPLEX*8, function is REAL*4. If argument is COMPLEX*16, function is REAL*8.
DIMAG(*cmp16*)	Imaginary part of COMPLEX*16 number	COMPLEX*16	REAL*8
CONJG(*cmp8*)	Conjugate of **COMPLEX*8** number	**COMPLEX*8**	**COMPLEX*8**
DCONJG(*cmp16*)	Conjugate of COMPLEX*16 number	COMPLEX*16	COMPLEX*16

Note: See Table 3.1 for a list of the abbreviations used in this table.

The intrinsic functions **AIMAG**, IMAG, and DIMAG return the imaginary part of complex numbers. The intrinsic functions **CONJG** and DCONJG return the complex conjugates of complex numbers. So, for a complex number *complex* equal to (*real*,*imag*), **AIMAG**(*complex*) equals *imag*, and **CONJG**(*complex*) equals (*real*, − *imag*). Note that the **REAL** and DBLE intrinsic functions, described in Section 3.11.3.1, can be used to return the real part of **COMPLEX*8** and COMPLEX*16 numbers, respectively.

■ Example

The following program uses complex intrinsic functions:

```
C This program applies the quadratic formula to a
C polynomial and allows for complex results.

      REAL A,B,C
      COMPLEX ANS1, ANS2, DESC
      WRITE (*,100)
100   FORMAT(' Enter a, b, and c of the
     +        polynomial aX^2 + bX + c:')
      READ (*,'(3F10.5)') A,B,C

      DESC = CSQRT(CMPLX(B*B-4.*A*C))
      ANS1 = (-B+DESC)/(2.*A)
      ANS2 = (-B-DESC)/(2.*A)

      WRITE (*,200)
200   FORMAT (/' The roots are:'/)
      WRITE (*,300) REAL(ANS1), AIMAG(ANS1),
     +              REAL(ANS2), AIMAG(ANS2)
300   FORMAT (' X = ',F10.5,'  +',F10.5,'i')
      STOP ' '
      END
```

3.11.3.9 Square Roots

Table 3.10 summarizes the intrinsic functions that return square roots.

Table 3.10
Intrinsic Functions: Square Roots

Name	Argument Type	Function Type
SQRT(*gen*)	real or cmp	Same as argument
DSQRT(*dbl*)	**REAL*8**	**REAL*8**
CSQRT(*cmp8*)	**COMPLEX*8**	**COMPLEX*8**
CDSQRT(*cmp16*)	COMPLEX*16	COMPLEX*16

Note: See Table 3.1 for a list of the abbreviations used in this table.

The intrinsic functions **SQRT**, **DSQRT**, **CSQRT**, and **CDSQRT** return the square root of their respective arguments. Note that the arguments to these intrinsic functions must be greater than or equal to zero. For a complex argument, **SQRT**, **CSQRT**, and **CDSQRT** return the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

■ Example

The following program uses the **SQRT** intrinsic function:

```
C This program calculates the length of the hypotenuse
C of a right triangle from the lengths of the other
C two sides.

        real sidea, sideb, hyp

        sidea = 3.
        sideb = 4.

        hyp = sqrt(sidea**2 + sideb**2)

        write(*,100) hyp
 100    format (/' The hypotenuse is ',f10.3)

        stop ' '
        end
```

3.11.3.10 Exponents and Logarithms

Table 3.11 lists the intrinsic functions that return exponents or logarithms.

Table 3.11
Intrinsic Functions: Exponents and Logarithms

Name	Definition	Argument Type	Function Type
EXP(*gen*)	Exponent	real or cmp	Same as argument
DEXP(*dbl*)	Exponent	**REAL*8**	**REAL*8**
CEXP(*cmp8*)	Exponent	**COMPLEX*8**	**COMPLEX*8**
CDEXP(*cmp16*)	Exponent	**COMPLEX*16**	**COMPLEX*16**
LOG(*gen*)	Natural logarithm	real or cmp	Same as argument
ALOG(*real4*)	Natural logarithm	**REAL*4**	**REAL*4**
DLOG(*dbl*)	Natural logarithm	**REAL*8**	**REAL*8**
CLOG(*cmp8*)	Natural logarithm	**COMPLEX*8**	**COMPLEX*8**
CDLOG(*cmp16*)	Natural logarithm	**COMPLEX*16**	**COMPLEX*16**
LOG10(*real*)	Common logarithm	real	Same as argument
ALOG10(*real4*)	Common logarithm	**REAL*4**	**REAL*4**
DLOG10(*dbl*)	Common logarithm	**REAL*8**	**REAL*8**

Note: See Table 3.1 for a list of the abbreviations used in this table.

The intrinsic functions **EXP**, **DEXP**, **CEXP**, and **CDEXP** return **e** * ***gen***.

The intrinsic functions **LOG**, **ALOG**, **DLOG**, **CLOG** and **CDLOG** return the natural logarithm of their respective arguments. **LOG10**, **ALOG10**, and **DLOG10** return the base-10 logarithm of their arguments.

For all of the logarithmic intrinsic functions, if the argument is real, the argument must be greater than zero. The complex argument (0,0) is not allowed in the intrinsic functions **CLOG** or **LOG**. The imaginary part of the result of **CLOG** is greater than $-\pi$ and less than or equal to π. The imaginary part of the result of **CLOG** is π only if the real part of the argument is less than zero and the imaginary part of the argument equals zero.

■ **Example**

The following program uses the **EXP** intrinsic function:

```
C Given the initial size and growth rate of a colony,
C this program computes the size of the colony at a
C specified time. The growth rate of the colony is
C assumed to be proportional to its size.

      REAL SIZEI, SIZEF, TIME, RATE

      SIZEI = 10000.
      TIME  = 40.5
      RATE  = 0.0875

      SIZEF = SIZEI * EXP(RATE * TIME)

      WRITE (*,100) SIZEF
100   FORMAT (' THE FINAL SIZE IS ',E12.6)

      STOP ' '
      END
```

3.11.3.11 Trigonometric Functions

Table 3.12 summarizes the trigonometric intrinsic functions.

Table 3.12
Intrinsic Functions: Trigonometric Functions

Name	Definition	Argument Type	Function Type
SIN(*gen*)	Sine	real or cmp	Same as argument
DSIN(*dbl*)	Sine	**REAL*8**	**REAL*8**
CSIN(*cmp8*)	Sine	**COMPLEX*8**	**COMPLEX*8**
CDSIN(*cmp16*)	Sine	**COMPLEX*16**	**COMPLEX*16**
COS(*gen*)	Cosine	real or cmp	Same as argument
DCOS(*dbl*)	Cosine	**REAL*8**	**REAL*8**
CCOS(*cmp8*)	Cosine	**COMPLEX*8**	**COMPLEX*8**
CDCOS(*cmp16*)	Cosine	**COMPLEX*16**	**COMPLEX*16**
TAN(*real*)	Tangent	real	Same as argument
DTAN(*dbl*)	Tangent	**REAL*8**	**REAL*8**
ASIN(*real*)	Arc sine	real	Same as argument
DASIN(*dbl*)	Arc sine	**REAL*8**	**REAL*8**
ACOS(*real*)	Arc cosine	real	Same as argument
DACOS(*dbl*)	Arc cosine	**REAL*8**	**REAL*8**
ATAN(*real*)	Arc tangent	real	Same as argument
DATAN(*dbl*)	Arc tangent	**REAL*8**	**REAL*8**
ATAN2(*realA*,*realB*)	Arc tangent (*realA*/*realB*)	real	Same as argument
DATAN2(*dblA*,*dblB*)	Arc tangent (*dblA*/*dblB*)	**REAL*8**	**REAL*8**
COTAN(*real*)	Cotangent	real	Same as argument
DCOTAN(*dbl*)	Cotangent	**REAL*8**	**REAL*8**
SINH(*real*)	Hyperbolic sine	real	Same as argument
DSINH(*dbl*)	Hyperbolic sine	**REAL*8**	**REAL*8**
COSH(*real*)	Hyperbolic cosine	real	Same as argument
DCOSH(*dbl*)	Hyperbolic cosine	**REAL*8**	**REAL*8**
TANH(*real*)	Hyperbolic tangent	real	Same as argument
DTANH(*dbl*)	Hyperbolic tangent	**REAL*8**	**REAL*8**

Note: See Table 3.1 for a list of the abbreviations used in this table.

All angles used in the trigonometric intrinsic functions are in radians. Table 3.13 indicates some restrictions on the arguments to and results of trigonometric intrinsic functions.

Table 3.13

Restrictions on Arguments and Results

Function	Restrictions on Arguments	Range of Results
SIN, DSIN, COS, DCOS, TAN, DTAN	None	
ASIN, DASIN	$\|arg\| <= 1$	$-\pi/2 <= result <= \pi/2$
ACOS, DACOS	$\|arg\| <= 1$	$0 <= result <= \pi$
ATAN, DATAN	None	$-\pi/2 <= result <= \pi/2$
ATAN2, DATAN2	Arguments cannot both be zero.	$-\pi <= result <= \pi$
COTAN	Argument cannot be zero.	

The range of the results of the intrinsic functions **ATAN2** and **DATAN2** is as follows:

Arguments	Result
$genA > 0$	$result > 0$
$genA = 0$ and $genB > 0$	$result = 0$
$genA = 0$ and $genB < 0$	$result = \pi$
$genA < 0$	$result < 0$
$genB = 0$	$\|result\| = \pi/2$

■ Example

The following program uses trigonometric intrinsic functions:

```
C This program asks for a polar coordinate and converts
C it to a rectangular coordinate.
C
      real theta, radius, x, y

      write (*,*) ' Enter polar coordinate(radius, angle)'
      read  (*,'(2f10.5)') radius, theta

      x = radius * cos(theta)
      y = radius * sin(theta)

      write (*,100) x,y
 100  format (/' (X,Y) =  (',f7.3,',',f7.3,')' )

      stop ' '
      end
```

3.11.3.12 Character Functions

Table 3.14 summarizes the intrinsic functions that perform operations on character constants and variables.

Table 3.14

Intrinsic Functions: Character Functions

Name	Definition	Argument Type	Function Type
LGE(*charA*,*charB*)	*charA* >= *charB*	char	log
LGT(*charA*,*charB*)	*charA* > *charB*	char	log
LLE(*charA*,*charB*)	*charA* <= *charB*	char	log
LLT(*charA*,*charB*)	*charA* < *charB*	char	log
LEN(*char*)	Length of string	char	int
INDEX(*charA*,*charB*)	Position of substring *charB* in string *charA*	char	int

Note: See Table 3.1 for a list of the abbreviations used in this table.

The intrinsic functions **LGE**, **LGT**, **LLE**, and **LLT** use the ASCII collating sequence to determine whether a character argument is less than (precedes in the ASCII collating sequence), greater than (follows in the ASCII collating sequence), or equal to (identical in the ASCII collating sequence) another character argument. If the lengths of two character arguments are not equal, the shorter operand is padded to the size of the larger operand by adding blanks on the right.

The argument to the **LEN** intrinsic function does not have to be defined.

The **INDEX** intrinsic function returns integers indicating the position of *charB* in *charA*. If the length of *charA* is less than the length of *charB*, or if *charB* does not occur in *charA*, the index equals zero. If *charB* occurs more than once in *charA*, the **INDEX** intrinsic function returns the position of the first occurrence of *charB*.

■ Example

The following list shows examples of the character intrinsic functions:

Function Reference	Equivalent
`LLE('A','B')`	`.TRUE.`
`LLT('A','a')`	`.TRUE.`
`LEN('abcdef')`	`6`
`INDEX('banana','an')`	`2`

3.11.3.13 End-of-File Function

Table 3.15 summarizes the end-of-file intrinsic function.

Table 3.15
Intrinsic Functions: End-of-File Function

Name	Definition	Argument Type	Function Type
EOF(*un*)	End-of-file	int	log

Note: See Table 3.1 for a list of the abbreviations used in this table.

If the unit specified by its argument is at or past the end-of-file record, the intrinsic function **EOF**(*int*) returns the value **.TRUE.**; otherwise, **EOF** returns the value **.FALSE.**. The value of *int* must be the unit specifier corresponding to an open file. The value of *int* cannot be zero, unless you have reconnected unit zero to a unit other than the screen or keyboard.

Example

The following program uses the **EOF** intrinsic function:

```
C This program reads a file of integers and
C prints their average.

      character*64  fname
      integer       total, count, value

      write(*,'(a)') ' Enter file name:  '
      read (*,'(a)') fname

C Open unit 1 for input (any unit except * is ok),
      open(1,file=fname)

      total = 0
      count = 0
  100 if (.not. eof(1)) then
        count = count + 1
        read(1,'(i7)') value
        total = total + value
        goto 100
      endif
      if (count .ge. 0) then
        write(*,*)'Average is:',float(total)/count
      else
        write(*,*) 'Input file is empty'
      endif
      stop ' '
      end
```

3.11.3.14 Address Functions

Table 3.16 lists the intrinsic functions that return addresses.

Table 3.16

Intrinsic Functions: Addresses

Name	Definition	Argument Type	Function Type
LOCNEAR(*var*)	Unsegmented address	Any	INTEGER*2
LOCFAR(*var*)	Segmented address	Any	INTEGER*4
LOC(*var*)	Address	Any	INTEGER*2 or INTEGER*4

Note: See Table 3.1 for a list of the abbreviations used in this table.

These three intrinsic functions return the machine address of the variable passed as an actual argument.

The following list shows how the address is returned for different types of arguments:

Argument	Return Value
Expression, function call, or constant	A temporary variable is generated to hold the result of the expression, function call, or constant. The address of the temporary variable is then returned.
All other arguments	The machine address of the actual argument is returned.

The value returned by a **LOCNEAR** intrinsic function is equivalent to a **near** procedure or data pointer in Microsoft C or an **adr** type in Microsoft Pascal. Similarly, the value returned by a **LOCFAR** intrinsic function is equivalent to a **far** data or function pointer in Microsoft C, or an **ads**, **adsfunc**, or **adsproc** type in Microsoft Pascal.

LOCNEAR can only be used with **NEAR** procedures and with objects in the default data segment, such as objects in **NEAR** common blocks and objects not named in **SLARGE** metacommands. For example, **LOCNEAR** will not usually work when applied to an argument unless that argument is explicitly in the default data segment. **LOC** returns either a near or a far pointer, depending on the memory model used to compile.

3.11.3.15 Bit-Manipulation Functions

Table 3.17 summarizes the functions that perform bit manipulation.

Table 3.17
Intrinsic Functions: Bit Manipulation

Name	Definition	Argument Type	Function Type
IOR(*intA*,*intB*)	Inclusive or	int	Same as argument
ISHL(*intA*,*intB*)	Logical shift	int	Same as argument
ISHFT(*intA*,*intB*)	Logical shift	int	Same as argument
ISHA(*intA*,*intB*)	Arithmetic shift	int	Same as argument
ISHC(*intA*,*intB*)	Rotate	int	Same as argument
IEOR(*intA*,*intB*)	Exclusive or	int	Same as argument
IAND(*intA*,*intB*)	Logical product	int	Same as argument
NOT(*intA*)	Logical complement	int	Same as argument
IBCLR(*intA*,*intB*)	Bit clear	int	Same as argument
IBSET(*intA*,*intB*)	Bit set	int	Same as argument
IBCHNG(*intA*,*intB*)	Bit change	int	Same as argument
BTEST(*intA*,*intB*)	Bit test	int	log

Note: See Table 3.1 for a list of the abbreviations used in this table.

All of the bit-manipulation intrinsic functions can be passed as actual arguments. These intrinsic functions work as follows:

Function	Operation
Inclusive or	All i bits of the result are set as follows: if the ith bit of either the first or second argument is 1, then the ith bit of the result is set to 1.
Logical shift	If *intB* is greater than or equal to zero, shift *intA* logically left by *intB* bits. If *intB* is less than zero, shift *intA* logically right by *intB* bits.

Arithmetic shift	If *intB* is greater than or equal to zero, shift *intA* arithmetically left by *intB* bits. If *intB* is less than zero, shift *intA* arithmetically right by *intB* bits.
Rotate	If *intB* is greater than or equal to zero, rotate *intA* left *intB* bits. If *intB* is less than zero, rotate *intA* right *intB* bits.
Exclusive or	If the *i*th bits of the first and second arguments are not equal to each other, then the *i*th bit of the result is set to 1. Otherwise, the *i*th bit of the result is set to 0.
Logical product	If the *i*th bits of both the first and the second arguments are 1, then the *i*th bit of the result is set to 1. Otherwise, the *i*th bit of the result is set to 0.
Logical complement	If the *i*th bit of the argument is 1 then the *i*th bit of the result is set to 0. Otherwise the *i*th bit of the result is set to 1.
Bit clear	Clear *intB* bit in *intA*.
Bit set	Set *intB* bit in *intA*.
Bit change	Reverse value of bit *intB* in *intA*.
Bit test	Return **.TRUE.** if bit *intB* in *intA* is set to 1. Return **.FALSE.** otherwise.

Examples

The following three examples show the results of three bit-manipulation intrinsic functions:

Function	Binary Representation	
IOR(240,90) = 250		11110000
	IOR	01011010
		11111010
IEOR(240,90) = 170		11110000
	IEOR	01011010
		10101010
IAND(240,90) = 80		11110000
	IAND	01011010
		01010000

Table 3.18 shows the results of other bit-manipulation intrinsic functions.

Table 3.18
Bit-Manipulation Examples

Function Reference	IntA	Result
ISHFT(IntA,2)	10010000 11000101	01000011 00010100
ISHFT(IntA,-2)	10010000 11000101	00100100 00110001
ISHA(IntA,3)	10000000 11011000	00000110 11000000
ISHA(IntA,-3)	10000000 11011000	11110000 00011011
ISHC(IntA,3)	01110000 00000100	10000000 00100011
ISHC(IntA,-3)	01110000 00000100	10001110 00000000
NOT(IntA)	00011100 01111000	11100011 10000111
IBCLR(IntA,4)	00011100 01111000	00011100 01101000
IBSET(IntA,14)	00011100 01111000	01011100 01111000
IBCHNG(IntA,5)	00011100 01111000	00011100 01011000
BTEST(IntA,2)	00011100 01111000	.FALSE.
BTEST(IntA,3)	00011100 01111000	.TRUE.

Chapter 4
The Input/Output (I/O) System

4.1 Overview

This chapter explains the input/output (I/O) system used in Microsoft FORTRAN. The following kinds of information are available:

For Information on:	See:
An overview of the input/output system	Section 4.2, "Introduction to the I/O System"
A summary of the I/O statements	Section 4.3, "I/O Statements"
Options available in I/O statements	Sections 4.3.1 – 4.3.11
Carriage-control characters	Section 4.7, "Carriage Control"
Formatted I/O	Section 4.8, "Formatted I/O"
Edit descriptors	Section 4.8.1, "Nonrepeatable Edit Descriptors," and Section 4.8.2, "Repeatable Edit Descriptors"
List-directed I/O	Section 4.9, "List-Directed I/O"

4.2 Introduction to the I/O System

In Microsoft FORTRAN's I/O system, data are stored in files and can be transferred between files. There are two basic types of files:

Type of File	Description
External files	Either a device, such as the screen, the keyboard, or a printer; or, a file that is stored on a device, such as a file on a disk.
Internal files	A character substring, character variable, character array element, character array, or noncharacter array. For information on internal files, see Section 4.6, "Internal Files."

All files are made up of records, which are sequences of characters or values. See the *Microsoft FORTRAN Compiler User's Guide* for information on the format of records in your operating system.

Input is the transfer of data from a file to internal storage. Output is the transfer of data from internal storage to a file. You input data by reading from a file, and output data by writing to a file.

4.3 I/O Statements

I/O statements in FORTRAN transfer or edit data, manipulate files, or determine or describe the properties of the connections to files. Table 4.1 lists the I/O statements.

Table 4.1

I/O Statements

Statement	Function
BACKSPACE	Positions a file back one record
CLOSE	Disconnects a unit
ENDFILE	Writes an end-of-file record
INQUIRE	Determines properties of a unit or named file
LOCKING	Controls access to information in specific direct-access files and/or records
OPEN	Associates a unit number with a file
PRINT	Outputs data to the asterisk (*) unit
READ	Inputs data
REWIND	Repositions a file to its initial point
WRITE	Outputs data

Note

In addition to the I/O statements, Microsoft FORTRAN includes an I/O intrinsic function, **EOF**(*unitspec*), which is described in Section 3.11.3.13, "End-of-File Function." **EOF** returns a logical value that indicates whether there are any data remaining in the file after the current position.

In I/O statements, you can specify a series of options. The following, for example, is the syntax of the **CLOSE** statement:

CLOSE (⟦**UNIT=**⟧*unitspec*
⟦**,ERR=***errlabel*⟧
⟦**,IOSTAT=***iocheck*⟧
⟦**,STATUS=***status*⟧)

The four options of the **CLOSE** statement are ⟦**UNIT=**⟧*unitspec*, ⟦**,ERR=***errlabel*⟧, ⟦**,IOSTAT=***iocheck*⟧, and ⟦**,STATUS=***status*⟧. The **UNIT=** option specifies the unit to be closed, the **ERR=** and **IOSTAT=** options allow you to control error handling, and the **STATUS=** option lets you specify whether to keep or delete the file after disconnecting. The following statement, for example, closes a file connected to unit 2, does not provide any special error handling, and uses the default value of the **STATUS=** option:

```
CLOSE(UNIT=2)
```

Options that are used in only one statement are described in the discussion of that statement in Section 5.3, "Statement Directory." Twelve of the options in I/O statements, however, are used in more than one I/O statement. These options are described in this chapter. Table 4.2 lists these options, the I/O statements they are used in, and the section that describes each.

Table 4.2
I/O Options

Option	I/O Statements	Section
ACCESS = *access*	**INQUIRE**, **OPEN**	4.3.3, "File Access Method"
editlist	**FORMAT**, **PRINT**, **READ**, **WRITE**	4.3.6, "The Edit List"
ERR = *errlabel*	All except **PRINT**	4.3.10, "Error and End-of-File Handling"
FILE = *file*	**INQUIRE**, **OPEN**	4.3.1, "File Names"
[**FMT** =]*formatspec*	**PRINT**, **READ**, **WRITE**	4.3.7, "Format Specifier"
FORM = *form*	**INQUIRE**, **OPEN**	4.3.4, "File Structure"
iolist	**PRINT**, **READ**, **WRITE**	4.3.8, "Input/Output List"
BLOCKSIZE = *blocksize*	INQUIRE, OPEN	4.3.9, "Input Output Buffer Size"
IOSTAT = *iocheck*	All except **PRINT**	4.3.10, "Error and End-of-File Handling"
MODE = *mode*	INQUIRE, OPEN	4.3.11, "File Sharing"
REC = *rec*	LOCKING, **READ**, **WRITE**	4.3.5, "Record Number"
SHARE = *share*	INQUIRE, OPEN	4.3.11, "File Sharing"
[**UNIT** =]*unitspec*	All except **PRINT**	4.3.2, "Units"

The following list briefly summarizes the options listed in Table 4.2. Sections 4.3.1 – 4.3.11 provide a complete discussion of these options.

Option	Description
ACCESS =	Specifies or determines the method of file access, which can be **'SEQUENTIAL'** or **'DIRECT'**.
BLOCKSIZE =	Specifies or determines the internal buffer size used in I/O.
editlist	Lists edit descriptors. It is used in **FORMAT** statements and format specifiers (the **FMT** = *formatspec* option) to describe the format of data.

ERR=	Controls I/O error handling. The **ERR=** option specifies the label of an executable statement.
IOSTAT=	Controls I/O error handling. The **IOSTAT=** option specifies a variable that is set to indicate whether an error has occurred.
FILE=	Specifies the name of a file.
FMT=	Specifies an *editlist* to use to format data.
FORM=	Specifies a file's format, which can be **'FORMATTED'**, **'UNFORMATTED'**, or 'BINARY'.
iolist	Specifies items to be input or output.
MODE=	Controls how other processes can access a file on networked systems. The **MODE=** option can be set to 'READWRITE', 'READ', or 'WRITE'.
SHARE=	Controls how other processes can simultaneously access a file on networked systems. The **SHARE=** option can be set to 'COMPAT', 'DENYNONE', 'DENYWR', 'DENYRD', or 'DENYRW'.
REC=	Specifies the first (or only) record of a file to be locked, read from, or written to.
UNIT=	Specifies the unit to which a file is connected.

4.3.1 File Names (FILE=)

A file can have a name. File names must follow the rules listed in Section 2.3, "Names." The name of an internal file is the name of the character substring, character variable, character array element, character array, or noncharacter array that makes up the file. The name of an external file is a character string identical to the name by which the file is known to the operating system. (Note that some operating systems are case insensitive.) External file names must obey the file-naming conventions of your operating system, as well as the rules listed in Section 2.3, "Names."

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

- If the file is opened with an **OPEN** statement, the name can be specified in the **OPEN** statement.
- If the file is opened with an **OPEN** statement and no name is specified in the **OPEN** statement, the file is considered a scratch or temporary file, and a default name is used. For the name used by your operating system, see the section on temporary scratch-file names in the *Microsoft FORTRAN Compiler User's Guide.*
- If the file is opened with an **OPEN** statement and the name is specified as all blanks, the name can be read from the command line or can be input by the user, as described in Section 5.3.38, "The OPEN Statement."
- If the file is referred to in a **READ** or **WRITE** statement before it has been opened, an implicit open operation is performed, as described in Sections 5.3.43, "The READ Statement," and 5.3.52, "The WRITE Statement." The implicit open operation is equivalent to executing an **OPEN** statement with a name specified as all blanks. Therefore the name is read from the command line or can be input by the user, as described in Section 5.3.38, "The OPEN Statement."

4.3.2 Units (UNIT=)

For most I/O operations, a file must be identified by a unit specifier. The unit specifier of an internal file is the same as the name of that internal file (see Section 4.6 for information on internal files). For an external file, you can connect a file to a unit specifier with the **OPEN** statement. Some external unit specifiers, listed below, are preconnected to certain devices and do not have to be opened. External units that you connect are disconnected when program execution terminates, or when the unit is closed by a **CLOSE** statement.

The unit specifier is required for all I/O statements except **PRINT** (because this always writes to standard output), **READ** with a format specifier and an I/O list only (because this always reads from standard output), and **INQUIRE** by file (because this specifies the file name, rather than the unit with which the file is associated).

An external unit specifier must be either an integer expression or an asterisk (*). The integer expression must be in the range −32,767 to −1, or 0 to 32,767. The following example connects the external file `undamp` to unit 10 and writes to it:

```
OPEN(UNIT=10,FILE='undamp')
WRITE (10,*) ' Undamped Motion:'
```

The asterisk (*) unit specifier is preconnected and cannot be connected by an **OPEN** statement. It is the only unit specifier that cannot be reconnected to another file. The asterisk unit specifier specifies the keyboard when reading and the screen when writing. The following example uses the asterisk unit specifier to write to the screen:

```
WRITE (*, '(1X,''Begin output.'')')
```

Microsoft FORTRAN has four preconnected external units:

External Unit	Description
Asterisk (*)	Always represents the keyboard and screen
0	Initially represents the keyboard and screen
5	Initially represents the keyboard
6	Initially represents the screen

The asterisk (*) unit cannot be connected to any other file, and attempting to close this unit produces an error during compilation. Units 0, 5, and 6, however, can be connected to any file by using the **OPEN** statement. If you close unit 0, 5, or 6, it is automatically reconnected to the keyboard and screen, the keyboard, or the screen, respectively.

If you read or write to a unit that has been closed, the file is opened implicitly, as described in Sections 5.3.43 and 5.3.52, "The READ Statement" and "The WRITE Statement."

■ Examples

In the following program, the character variable `fname` is an internal file:

```
C The output of this program is
C FM004.DAT
      CHARACTER*14 fname
      ifil=4
      WRITE (fname,100)ifil
100   FORMAT('FM',I3.3,'.DAT')
      WRITE(*,*)fname
      END
```

The following example writes to the preconnected unit 6 (the screen), then reconnects unit 6 to an external file and writes to it, and finally reconnects unit 6 to the screen and writes to it.

```
      REAL a,b
C Write to the screen (preconnected unit 6)
      WRITE(6,'('' This is unit 6'')')
C Use the OPEN statement to connect unit 6 to an
C external file named COSINES
      OPEN(UNIT=2*3,FILE='COSINES',STATUS='NEW')
      DO 200 a=.1,6.3,.1
         b=COS(a)
C Write to the file COSINES
         WRITE(6,100)a,b
100      FORMAT(F3.1,F5.2)
200   CONTINUE
C Reconnect unit 6 to the screen, by closing it
      CLOSE(6,STATUS='KEEP')
C Write to the screen
      WRITE(6,'('' Cosines completed'')')
      STOP ' '
      END
```

4.3.3 File Access Method (ACCESS=)

The following sections describe the two methods of file access: sequential and direct.

4.3.3.1 Sequential File Access

Sequential files contain records whose order is the sequential order in which the records were written.

All internal files use sequential access. You must also use sequential access for files associated with "sequential devices." A sequential device is a device that does not allow explicit motion (other than reading or writing). The keyboard, screen, and printer are all sequential devices.

Note that direct operations to files opened for sequential access are not allowed.

4.3.3.2 Direct File Access

Direct files are random-access files whose records can be read or written in any order. Direct-access files must reside on disk. You cannot associate a direct-access file with a sequential device, such as the keyboard, screen, or printer.

Records are numbered sequentially, with the first record numbered 1. All records have the same length, specified by the **RECL=** option in the **OPEN** statement. Except for binary files, the number of bytes written to a record must be less than or equal to the record length. One record is written for each unformatted **READ** or **WRITE** statement. A formatted **READ** or **WRITE** statement can transfer more than one record using the slash (**/**) edit descriptor. For binary files, a single READ or WRITE statement can read or write as many records as required to accommodate the number of bytes being transferred. On output, incomplete formatted records are padded with spaces. Incomplete unformatted and binary records are padded with undefined bytes (zeros).

In a direct-access file, it is possible to write records out of order (for example, 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 is illegal and can produce a run-time error. If a record is written beyond the old terminating file boundary, the operating system attempts to extend direct-access files. This works only if there is room available on the physical device.

Note that sequential operations to files opened for direct access are allowed. The operation takes place on the next record. The following statements, for example, read the third and fourth records of the file xxx:

```
.
.
.
OPEN(1,FILE='xxx',ACCESS='DIRECT',RECL=15
+  FORM='FORMATTED')
READ(1,'(3I5)',REC=3)i,j,k
READ(1,'(3I5)')l,m,n
.
.
.
```

4.3.4 File Structure (FORM=)

The structure of a file depends on two things:

1. The access to the file, which is set by the **ACCESS=** option described in Section 4.3.3
2. The form of the data in the file

A file is structured in one of the following three ways:

Form	Description of Structure
Formatted	A file is a sequence of formatted records. Formatted records are a series of ASCII characters terminated by an end-of-record mark. The records in a formatted sequential file can have varying lengths. All the records in a formatted direct file must be the same length. All internal files must be formatted.
Unformatted	A file is a sequence of unformatted records. Unformatted records are a sequence of values. Unformatted direct files contain only this data, and each record is padded to a fixed length with undefined bytes. Unformatted sequential files also contain

information that indicates the boundaries of each record.

Binary

Binary sequential files are sequences of bytes with no internal structure. There are no records. The file contains only the information specified as I/O list items in **WRITE** statements referring to the file.

Binary direct files have very little structure. A record length is assigned by the **RECL=** option of the **OPEN** statement. This establishes record boundaries, which are used only for repositioning and padding before and after read and write operations and during **BACKSPACE** operations. These record boundaries do not, however, restrict the number of bytes that can be transferred during a read or write operation. If an I/O operation attempts to read or write more values than are contained in a record, the read or write operation is continued on the next record. Note that I/O operations that can be performed on unformatted direct files produce the same results when applied to binary direct files.

In both sequential and direct binary files, data read must correspond in position to data previously written.

See the *Microsoft FORTRAN Compiler User's Guide* for information on how records are represented on your system.

4.3.5 Record Number (REC=)

The **REC=** option is used to specify the number of a specific record. In the LOCKING statement, *rec* specifies the first record to be locked or unlocked. In the **READ** and **WRITE** statements, *rec* specifies the first record to be read or written. The first record in a file is record number 1.

4.3.6 The Edit List

Edit lists describe the format of data. They are used in **FORMAT** statements and format specifiers.

The edit list is a list of editing descriptions separated by commas. You may omit the comma between two list items if the resulting edit list is

not ambiguous. For example, you can usually omit the comma after a P edit descriptor or before or after the slash (/) edit descriptor without ambiguity.

There are three types of editing descriptions:

1. Nonrepeatable edit descriptors (*nonrepeatable*)
2. Repeatable edit descriptors optionally preceded by a repeat specification (*nrepeatable*)
3. An edit list, enclosed in parentheses, optionally preceded by a repeat specification (*n*(*editlist*))

A repeat specification is a nonzero-unsigned-integer constant. Edit descriptors are described in Sections 4.8.1, "Nonrepeatable Edit Descriptors," and 4.8.2, "Repeatable Edit Descriptors." The following list gives examples of each type of editing description:

Editing Description	Examples
nonrepeatable	`'Total = '`
	`3Hyes`
	`SP`
	`BN`
	`1X`
nrepeatable	`2I5`
	`I5`
	`10F8.2`
	`3A10`
	`E12.7E2`
n(*editlist*)	`2(1X,2I5)`
	`(1X, 'Total = ', E12.7E2)`
	`(3A10, 10F8.2)`
	`920(10F8.2)`
	`2(13(2I5),SP,'Total =',F8.2)`

Up to 16 levels of nested parentheses are permitted within the outermost level of parentheses in an *editlist*.

■ Examples

The following program contains two examples of *editlist*:

```
      INTEGER a,b
      REAL x,y
      DATA a /52/, b /9/, x /5832.67/, y /1.02781/
      WRITE(*,100)a,b
C The editlist in the following FORMAT statement is below the
C hyphens:   --------------------
100   FORMAT(' A = ',I5,'B = ',I5)
      WRITE(*,200)x,y
C The editlist in the following FORMAT statement is below the
C hyphens:   ---------------
200   FORMAT(' ',2(F8.3,1Hm))
      STOP ' '
      END
```

This program writes the following on the screen:

```
A =     52B =      9
5832.670m    1.028m
```

Note that each formatted **WRITE** statement writes a blank to the terminal device as a carriage-control character (described in Section 4.7).

4.3.7 Format Specifier (FMT=)

Format specifiers either contain an edit list or indicate what edit list to use to determine the format of data. Format specifiers are used in **PRINT**, **READ**, and **WRITE** statements.

The following sections show the seven acceptable types of format specifiers and a generalized example of how each format specifier could be used.

4.3.7.1 FORMAT Statement Label

A format specifier can be the label of a **FORMAT** statement. In this case, the edit list specified in the **FORMAT** statement is used to format the data to be input or output. The following generalized example shows how a **FORMAT** statement label can be used to specify an edit list for a **WRITE** statement:

```
      WRITE (*,label) iolist
label FORMAT(editlist)
```

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

4.3.7.2 Integer-Variable Name

An **ASSIGN** statement can be used to associate an integer variable with the label of a **FORMAT** statement. The integer variable can then be used as a format specifier, as follows:

```
      ASSIGN label TO var
label FORMAT(editlist)
      WRITE(*,var) iolist
```

In the **WRITE** statement, the integer-variable name *var* refers to the **FORMAT** statement *label*, as assigned just before the **FORMAT** statement. For further information, see Section 5.3.1, "The ASSIGN Statement."

4.3.7.3 Character Expression

An edit list can be written as a character expression, and that character expression can be used as a format specifier, as follows:

```
WRITE(*,'(editlist)')iolist
```

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

The character expression can be a character constant. It cannot be an expression involving concatenation of an operand whose length specifier is an asterisk in parentheses, unless that operand is the symbolic name of a constant.

4.3.7.4 Character Variable

An edit list can be written as a character expression, and that expression can be assigned to a character variable. The character variable is then used as the format specifier, as follows:

```
CHARACTER*n var
var = '(editlist)'
WRITE(*,var)iolist
```

In this example, the **WRITE** statement uses the character variable *var* as the format specifier for data transfer.

4.3.7.5 Asterisk (*)

When an asterisk (*) is used as a format specifier, list-directed I/O is performed, as follows:

WRITE(*,*) *iolist*

In this statement, the second asterisk indicates a list-directed I/O transfer. For more information, see Section 4.9, "List-Directed I/O."

4.3.7.6 Character or Noncharacter Array Name

An edit list can be written as a character expression, and that expression can be assigned to an array. The array is then used as the format specifier, as follows:

CHARACTER **bytes* *array*(*dim*)
DATA *array* **/'**(*editlist*)**'/** ...
WRITE(*,*array*)*iolist*

The array is interpreted as all of the elements of the array concatenated in column-major order (see Section 5.3.12, "The DIMENSION Statement," for information on order of array elements). If a noncharacter array is specified, the elements of the array are treated as equivalent character variables of the same length.

The edit list of a Hollerith format specifier cannot contain an apostrophe edit descriptor or an H edit descriptor.

■ Example

The following program uses a character array, CA, and a real array, RA, to write THE TEST SUCCEEDED twice:

```
$NOTSTRICT
      CHARACTER*8 CA(4)
      REAL*8 RA(4)
      DATA CA/'(19H THE',' TEST SU','CCEEDED)','     '/
      DATA RA/'(19H THE',' TEST SU','CCEEDED)','   '/
      WRITE(*,CA)
      WRITE(*,RA)
      END
```

4.3.7.7 Character Array Element

An edit list can be written as a character expression, and that expression can be assigned to a character array element. The character array element is then used as the format specifier, as follows:

```
CHARACTER*bytes array(dim)
array(i) = '(editlist)'
WRITE(*,array(i))iolist
```

In this example, the **WRITE** statement uses the character array element *array(i)* as the format specifier for data transfer.

4.3.8 Input/Output List

The input/output list, *iolist*, specifies the items to input or output. See Section 4.8.3 for an explanation of how the *iolist* and *editlist* interact.

The following items can be in an *iolist*:

Item	Notes

Nothing	An *iolist* can be empty. The resulting record is either of zero length or consists only of padding characters.
	`C Example of empty iolist`
	`      WRITE(2,100)`
	`100   FORMAT(' Powys, Puyallup')`

A variable name, an array-element name, or a character-substring name	These elements specify that the variable, array element, or character substring should be input or output.
	`C Example of variable and array`
	`C element in iolist`
	`      READ(*,500)webb,bahb(webb)`

An array name

An array name specifies all of the elements of the array, in column-major order. See Section 5.3.12, "The DIMENSION Statement," for an explanation of how arrays are arranged in memory.

```
C Example of array in iolist
C (writes      0     0     0     0     0)
      INTEGER handle(5)
      DATA handle /5*0/
      WRITE(*,99)handle
99    FORMAT(1X,5I5)
```

Any expression

Output lists in **WRITE** and **PRINT** statements can contain arbitrary expressions.

An implied-**DO** list

Implied-**DO** lists have the following form:

(*iolist*, *dovar*=*start*,*stop*[,*inc*])

Here, *iolist* is an input/output list (and can contain any of the items in this list, including another implied-**DO** list). The other variables are as described for implied-**DO** lists in Section 5.3.11, "The DATA Statement."

Items in an implied-**DO** list are analogous to statements within an ordinary **DO** loop. If an implied-**DO** list and a **DO** loop have the same values of *dovar*, *start*, *stop*, and *inc*, then the items in the implied-**DO** list are read or written in the same order as the statements within the **DO** loop are executed.

In a **READ** statement, the **DO** variable *dovar* (or anything associated with *dovar*) must not appear in the *iolist* in the implied-**DO** list. The variable *dovar* can, however, have been read in the same **READ** statement before the implied-**DO** list.

```
C Examples of input and output with
C implied-do list in iolist
      INTEGER c, handle(10),i
      WRITE(*,*)' Enter c (<11) and'
      WRITE(*,*)' handle(1) to handle(c)'
      READ(*,09)c,(handle(i),i=1,c)
09    FORMAT(I5,10(:,/,I5))
      WRITE(*,99)c,(handle(i),i=1,c)
99    FORMAT(1X,2I5)
      STOP ' '
      END
```

4.3.9 Input/Output Buffer Size (BLOCKSIZE=)

One way to control program size and speed at execution time is through the use of the **BLOCKSIZE=** option in the **OPEN** statement. The value of **BLOCKSIZE** is an integer expression specifying the internal buffer size for use in I/O. Because the specified buffer size is usually rounded up to the next block size, the **BLOCKSIZE=** option in the **INQUIRE** statement allows you to determine the actual buffer size used.

See the *Microsoft FORTRAN Compiler User's Guide* for specific information about block sizes and using the **BLOCKSIZE=** option with your operating system.

4.3.10 Error and End-of-File Handling (IOSTAT=, ERR=, END=)

If an error or end-of-file record is encountered during the processing of an I/O statement, the action taken depends on the presence and definition of the **ERR=***errlabel*, **IOSTAT=***iocheck*, and **END=***endlabel* options. Note that encountering an end-of-file record is treated as an error by every I/O statement except **READ**.

Since the **PRINT** statement does not allow any of these options to be specified, an error that occurs during execution of a **PRINT** statement always produces a run-time error.

Table 4.3 indicates what action is taken when an error or end-of-file record is encountered by a **READ** statement. Note that any time an error occurs during a **READ** statement, all the items in *iolist* become undefined.

Table 4.3

Errors and End-of-File Records When Reading

IOSTAT Set	END Set	ERR Set	End-of-File Occurs	Error, or Error and End-of-File, Occurs
No	No	No	Run-time error is produced.	Run-time error is produced.
No	No	Yes	Go to *errlabel*.	Go to *errlabel*.
No	Yes	No	Go to *endlabel*.	Run-time error is produced.
No	Yes	Yes	Go to *endlabel*.	Go to *errlabel*.
Yes	No	No	Set *iocheck* negative and continue.	Set *iocheck* positive and continue.
Yes	No	Yes	Set *iocheck* negative and continue.	Set *iocheck* positive and go to *errlabel*.
Yes	Yes	No	Set *iocheck* negative and go to *endlabel*.	Set *iocheck* positive and continue.
Yes	Yes	Yes	Set *iocheck* negative and go to *endlabel*.	Set *iocheck* positive and go to *errlabel*.

The following list shows what happens when an error (including encountering an end-of-file record) occurs during any I/O statement other than **READ** or **PRINT**:

If:	Then:
Neither *errlabel* nor *iocheck* is present	The program is terminated, with a run-time error message.
Only *errlabel* is present	Control is transferred to the statement at *errlabel*.
Only *iocheck* is present	The value of *iocheck* is set to a positive integer and control returns as if the statement had executed without error.
Both *errlabel* and *iocheck* are present	The value of *iocheck* is set positive and control is transferred to the statement at *errlabel*.

If an I/O statement terminates without encountering an error or end-of-file record, and *iocheck* is specified, *iocheck* is set to zero.

■ Examples

In the following program, none of the available options **ERR=**, **IOSTAT=**, or **END=** are set, so if an invalid value is entered for `i`, a run-time error is produced:

```
      INTEGER i
      WRITE(*,*)'Please enter i'
      READ(*,*)i
      WRITE(*,*)'This is i:',i
      STOP ' '
      END
```

The following program uses the **ERR=** option to prompt the user to enter a valid number:

```
      INTEGER i
      WRITE(*,*)'Please enter i:'
50    READ(*,*,ERR=100)i
      WRITE(*,*)'This is i:',i
      STOP ' '
100   CONTINUE
      WRITE(*,*)'Invalid value. Please enter new i:'
      GOTO 50
      END
```

The following program uses both the **ERR=** and **IOSTAT=** options to handle invalid input:

```
      INTEGER i,j
      DATA j /0/
      WRITE(*,*)'Please enter i:'
50    READ(*,*,ERR=100,IOSTAT=j)i
      WRITE(*,*)'This is i:',i,' iostat = ',j
      STOP ' '
100   CONTINUE
      WRITE(*,*)'iostat = ',j,' Please enter new i:'
      GOTO 50
      END
```

4.3.11 File Sharing (MODE=, SHARE=)

In systems that allow multitasking or use networking, more than one program can attempt to access the same file at the same time. Two options (**MODE=** and **SHARE=**) in the **OPEN** statement allow you to control access to files. These options are also in the **INQUIRE** statement, so you can determine the access status of a file.

The Value of:	Determines:
MODE=	How the first process to open a file can use that file. You can choose to be able to read the file (**'READ'**), write to the file (**'WRITE'**), or do both (**'READWRITE'**).
SHARE=	How subsequent processes are allowed to access the file (while that file is still open). You can choose to allow subsequent processes to read the file (**'DENYWR'**), write to the file (**'DENYRD'**), do both (**'DENYNONE'**), or do neither (**'DENYRW'**). You can also choose not to allow any processes, except the process that originally opened the file, to open the file (**'COMPAT'**).

Table 4-4, "Mode and Share Values," indicates the restrictions on opening a file that has already been opened with a particular value of *mode* and *share*.

Table 4.4
Mode and Share Values

Original Process Opened with:		Concurrent Processes Can Be Opened with:	
MODE=	And SHARE=	MODE=	And SHARE=
'READWRITE' or 'READ' or 'WRITE'	'COMPAT'	'READWRITE' or 'READ' or 'WRITE'	'COMPAT' by original process only
'READWRITE' or 'READ' or 'WRITE'	'DENYRW'	Cannot be concurrently opened	
'READWRITE'	'DENYWR'	'READ'	'DENYNONE'
'READ'	'DENYWR'	'READ'	'DENYNONE' or 'DENYWR'
'WRITE'	'DENYWR'	'READ'	'DENYNONE' or 'DENYRD'
'READWRITE'	'DENYRD'	'WRITE'	'DENYNONE'
'READ'	'DENYRD'	'WRITE'	'DENYNONE' or 'DENYWR'
'WRITE'	'DENYRD'	'WRITE'	'DENYNONE' or 'DENYRD'
'READWRITE'	'DENYNONE'	'READWRITE' or 'READ' or 'WRITE'	'DENYNONE'
'READ'	'DENYNONE'	'READWRITE' or 'READ' or 'WRITE'	'DENYNONE' or 'DENYWR'
'WRITE'	'DENYNONE'	'READWRITE' or 'READ' or 'WRITE'	'DENYNONE' or 'DENYRD'

If, for example, you have a file that is opened with **MODE='READ'** and **SHARE='DENYRD'**, that file can also be opened with **MODE='WRITE'** and **SHARE='DENYNONE'** or **SHARE='DENYWR'**.

Suppose, for example, that you want several processes to read a file, and you want to ensure that no process updates the file while those processes are reading it. First, determine what type of access to the file you want to allow the original process. In this case, you want the original process only to read the file. Therefore, the original process should open the file with **MODE='READ'**. Now, determine what type of access the original process should allow other processes: in this case, other processes should only be able to read the file. Therefore, the first process should open the file with **SHARE='DENYWR'**. Now, as indicated in Table 4.4, other processes can also open the same file with **MODE='READ'** and **SHARE='DENYWR'**.

4.4 Choosing File Types

You can combine the available file properties in many ways to create many kinds of files. Two common file types are the following:

1. Sequential, formatted files associated with the asterisk (*) unit (which represents the keyboard and screen).

   Note that, when reading from the asterisk (*) unit (the keyboard), you must terminate lines by pressing ENTER. To correct typing mistakes, follow the conventions of your operating system.

2. A named, sequential, formatted external file.

■ Example

The following example uses the two types of files described above:

```
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 FROM EXTERNAL FILE; ANY NUMBER
C WILL DO.
      OPEN(3,FILE=FNAME)

C USE UNIT 4 FOR OUTPUT TO SECOND EXTERNAL FILE; ANY
C NUMBER EXCEPT 3 WILL DO.
      OPEN(4,FILE='OUT.TXT',STATUS='UNKNOWN')

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
```

The type of file you use depends on your application. The following list indicates some reasons for choosing other types of files:

If:	Then:
You need random access I/O	You must use direct-access files. A common example of this type of application is a data base. The form can be binary, formatted, or unformatted.

Your data are accessed only by Microsoft FORTRAN, and speed is important	Accessing binary and unformatted files is faster than accessing formatted files.
Data must be transferred without any system interpretation	Binary or unformatted I/O is most practical.
All 256 possible byte values (ASCII 0 – 255) are to be transferred	Binary or unformatted I/O is necessary.
You are controlling a device with a one-byte (binary) interface	Binary or unformatted I/O is necessary. In this situation, formatted I/O would interpret certain characters (such as the ASCII representation for **RETURN**) instead of passing them through to the program unaltered. The binary format may be preferable, since no record structure information is contained in the file.
Data must be transferred without any system interpretation, and will be read by non-FORTRAN programs	The binary format is recommended. Unformatted files are blocked internally, so the non-FORTRAN program must be compatible with this format to interpret the data correctly. Binary files contain only the data written to them.
You are reading a file that was not created by a Microsoft FORTRAN program	Binary I/O is recommended. Non-FORTRAN files usually have a different structure than FORTRAN files. A binary direct file opened with REC = 1, for example, is very similar to a stream file in the Microsoft C library. You can move to a position in the file and read an arbitrary sequence of values. Incomplete records don't cause undefined values because the record size is 1.

4.5 File Position

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

Executing the **ENDFILE** statement, or executing a **READ** statement at the end of the file, positions the file beyond the end-of-file record and produces an error. Attempting to read the end-of-file record or past it produces an error unless you have specified the **END=** option in a **READ** statement.

4.6 Internal Files

Most FORTRAN files are external; that is, they are a physical device, or they are a file that is known to the operating system. An internal file is a character substring, character variable, character array element, character array, or noncharacter array. You must follow these rules when using internal files:

- Use only formatted, sequential I/O.
- Use only the **READ** and **WRITE** I/O statements to refer to an internal file.
- Don't use list-directed formatting.

There are 2 basic types of internal files:

Type of File	Properties
Character variable, character array element, character substring, or noncharacter array	The file has exactly one record, which is the same length as the variable, array element, substring, or noncharacter array. Noncharacter arrays are allowed for compatibility with older versions of FORTRAN.
Character array	The file is a sequence of character array elements, each of which is a record. The order of records is the same as the order of array elements. All records are the same length: the length of array elements.

If less than an entire record is written to an internal file, the rest of the record is filled with blanks. Before an I/O statement is executed, internal files are positioned at the beginning of the file.

With internal files, you can use the formatting capabilities of the I/O system to convert values between external character representations and Microsoft 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 representations.

Note

The FORTRAN 66 **DECODE** statement has been replaced by the internal **READ** function. The **ENCODE** statement has been replaced by the internal **WRITE** function.

■ Example

```
C This example prompts for a 3-digit file id
C and uses an internal file to create a file name
      CHARACTER fname*64
      fname='FILE    .DAT'
      WRITE(*,*)'Enter 1-3 digit file identifier'
      READ(*,*)id
      WRITE(fname(5:7),100)id
100   FORMAT(I3.3)
      OPEN(1,file=fname,STATUS='NEW')
      .
      .
      .
      END
```

4.7 Carriage Control

When a record is transferred to a terminal device, such as the screen or a printer, using formatted I/O, the first character of that record is interpreted as a carriage-control character, and is not printed. The characters 0, 1, +, and the blank character, have the effects indicated in Table 4.5. Any other character is treated like the blank character. If the first character of your record does not print, make sure it is not being interpreted as the carriage-control character. In the following program fragment, for example, the number 2 is interpreted as a carriage-control character, and is treated like the space character:

```
      WRITE(*,100)
100   FORMAT('25 years')
```

The fragment above produces the following output:

```
5 years
```

Table 4.5

Carriage-Control Characters

Character	Effect
Blank	Advances one line.
0	Advances two lines.
1	Advances to top of next page. The screen behaves as if this carriage-control character were ignored.[a]
+	Does not advance. This can be used for overprinting.

[a] When a 1 is sent to the screen as a carriage-control character, the program emits an ASCII form-feed character, a backspace, a blank, and a carriage return. This is because the form-feed character would otherwise appear as a graphic character on the screen. The effect is that the character is ignored.

Note that the first character is *not* treated as a carriage-control character in list-directed I/O.

When writing terminal files, the end-of-record mark is normally not emitted until the next record is written. However, if a write operation to the screen is followed by a read operation from the keyboard, a new-line character is automatically emitted and the input line is positioned below the output line. To suppress the new-line character and display the input on the same line as the previous output, use the backslash (\) edit descriptor in the WRITE statement. The input appears at the end of the last line written. Since input lines always end with a new-line character, the next write operation always begins on a new line. Therefore, if the next operation to the console is a write operation, carriage control is adjusted to write one less end-of-record mark.

Note that the plus (+) carriage-control character has no effect if the previous console operation was a read.

4.8 Formatted I/O

If a **READ** or **WRITE** statement includes a format specifier, the I/O statement is a formatted I/O statement. The remainder of this section discusses the elements of format specifiers, and the interaction between format specifiers and the I/O list. See Section 4.3.7 for information on format specifiers.

4.8.1 Nonrepeatable Edit Descriptors

Table 4.6 summarizes the nonrepeatable edit descriptors.

Table 4.6
Nonrepeatable Edit Descriptors

Form	Name	Use	Used for Input	Used for Output
string	Apostrophe editing	Transmits *string* to output unit	No	Yes
*n***H**	Hollerith editing	Transmits next *n* characters to output unit	No	Yes
T*c* **TL***c* **TR***c*	Positional editing	Specifies position in record	Yes	Yes
*n***X**	Positional editing	Specifies position in record	Yes	Yes
SP **SS** **S**	Optional-plus editing	Controls output of plus signs	No	Yes
/	Slash editing	Positions to next record or writes end-of-record mark	Yes	Yes
\	Backslash editing	Continues same record	No	Yes (to terminal and printer only)
:	Format control termination	If no more items in *iolist*, terminates statement	No	Yes
*k***P**	Scale-factor editing	Sets scale for exponents in subsequent **F** and **E** (repeatable) edit descriptors	Yes	Yes
BN **BZ**	Blank interpretation	Specifies interpretation of blanks in numeric fields	Yes	No

Sections 4.8.1.1 – 4.8.1.10 describe each nonrepeatable edit descriptor.

4.8.1.1 Apostrophe Editing

- **Syntax**

string

If a format specifier contains a character constant, *string*, that *string* is transmitted to the output unit. Embedded blanks are significant; two adjacent apostrophes (that is, single quotation marks) must be used to represent a single apostrophe within a character constant. Apostrophe editing cannot be used with **READ** statements.

```
C These write statements both output ABC'DEF
C (The leading blank is a carriage-control character)
      WRITE (*,970)
970   FORMAT (' ABC''DEF')
      WRITE (*,'('' ABC''''DEF'')')
C The following write also outputs ABC'DEF. No carriage-
C control character is necessary for list-directed I/O
      WRITE(*,*)'ABC''DEF'
```

4.8.1.2 Hollerith Editing (H)

- **Syntax**

*n***H**

The *n***H** edit descriptor transmits the next *n* characters, with blanks counted as significant, to the output unit. Hollerith editing can be used in every context where constants occur.

The *n* characters transmitted are called a "Hollerith constant."

```
C These write statements both output ABC'DEF
C (The leading blank is a carriage-control character)
      WRITE (*,'(8H ABC''DEF)')
      WRITE (*,960)
960   FORMAT (8H ABC'DEF)
```

4.8.1.3 Positional Editing: Tab, Tab Left, Tab Right (T, TL, TR)

■ Syntax

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. Note that moving the position backward more than 512 bytes (characters) is not recommended.

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 *c* characters beyond the current position. The **TL***c* edit descriptor specifies that the transmission of the next character is to occur *c* characters prior to the current position.

If **TL***c* specifies a position before the first position of the record, **TL***c* editing causes transmission to or from position 1 of the current record.

If you use these edit descriptors to move to a new position that is to the right of the last data item transmitted, and a further data item is written, the space between the end of data in the record and the new position is filled with spaces. For example, consider the following program:

```
        WRITE(*,100) 5,9
100     FORMAT (I5,20X,TL10,I5)
        END
```

This program produces the following output (each x represents a space):

```
xxxx5xxxxxxxxxxxxxxx9
```

Be careful when using these edit descriptors if you read data from files that use commas as field delimiters. If you move backward in a record by using **TL***c* or by using **T***c* where *c* is less than the current position in the record, commas are disabled as field delimiters. If the format controller encounters a comma after you have moved backward in a record with **TL***c* or **T***c*, a run-time error is produced. If you want to move backward in a record without disabling commas as field delimiters, you can continue to the end-of-record mark, and then use the **BACKSPACE** statement to move to the beginning of the record.

4.8.1.4 Positional Editing (X)

- **Syntax**

*n*X

The *n*X edit descriptor advances the file position *n* characters. On output, if the *n*X edit descriptor moves past the end of data in the record, and if there are further items in the *iolist*, blanks are output, as described for the T*c* and TR*c* edit descriptors.

```
C This writes 1    5    10    15 on the screen
      WRITE(*,100)
100   FORMAT(1X,'1',3X,'5',3X,'10',3X,'15')
C This writes zogoZOGozogo      ! on the screen
      WRITE(*,10)
10    FORMAT(1X,'zogozogozogo',TL8,'ZOG',10X,'!')
```

4.8.1.5 Optional-Plus Editing (SP, SS, 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 **SS**, the default.

```
C The following statements write
C  251 +251  251 +251  251
      INTEGER i
      i = 251
      WRITE(*,100)i,i,i,i,i
100   FORMAT(1X,I5,SP,I5,SS,I5,SP,I5,S,I5)

C The following statements write
C  .673E+4+.673E+4 .673E+4+.673E+4 .673E+4
      REAL r
      r = 67.3E2
      WRITE(*,100)r,r,r,r,r
100   FORMAT(1X,E8.3E1,SP,E8.3E1,SS,E8.3E1,SP,E8.3E1,S,E8.3E1)
```

4.8.1.6 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 mark is written, and the file is positioned to write on the beginning of the next record.

```
C The following statements write a column and a row
      WRITE(*,100)
100   FORMAT(' c row',/,' o',/,' l',/,' u',/,' m',/,' n')
```

The output of the example above is:

```
c row
o
l
u
m
n
```

4.8.1.7 Backslash Editing (\)

The backslash edit descriptor is used only for formatted output to terminal devices, such as the screen or a printer. It is ignored in all other situations.

Normally when the format controller terminates, an end-of-record mark is written. If the last edit descriptor encountered by the format controller is a backslash (\), no end-of-record mark is written, so subsequent I/O statements can continue writing to the same record.

This mechanism can be used to write a prompt to the screen and then read a response from the same line, as in the following example:

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

4.8.1.8 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*.

```
C The following example writes
C a= 3.20 b=  .99
      REAL a,b,c,d
      DATA a /3.2/, b /.9871515/
      WRITE(*,100)a,b
100   FORMAT(' a=',F5.2,:,' b=',F5.2,
     +      :,' c=',F5.2,:,' d=',F5.2)
      STOP ' '
      END
```

4.8.1.9 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 is encountered. 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:

- On input, with **F** and **E** editing (if no explicit exponent exists in the field), and on output, with **F** editing, the externally represented number equals the internally represented number multiplied by 10^k.
- On input, with **F** and **E** editing, the scale factor has no effect if there is an explicit exponent in the input field.
- 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).

■ Examples

The following program fragment uses scale-factor editing when reading:

```
      read (*,100) a,b,c,d
100   format(f10.6, 1p, f10.6, f10.6, -2p, f10.6)

      write (*,200) a,b,c,d
200   format(4f11.3)
      end
```

Assume that the following data are entered:

```
 12340000  12340000  12340000  12340000
    12.34     12.34     12.34     12.34
  12.34e0   12.34e0   12.34e0   12.34e0
  12.34e3   12.34e3   12.34e3   12.34e3
```

The program outputs the following:

```
    12.340      1.234      1.234   1234.000
    12.340      1.234      1.234   1234.000
    12.340     12.340     12.340     12.340
12340.000  12340.000  12340.000  12340.000
```

The following program fragment uses scale-factor editing when writing:

```
      a = 12.34

      write (*,100) a,a,a,a,a,a
100   format(1x,f9.4,e11.4e2,1p,f9.4,e11.4e2,-2p,f9.4,e11.4e2)
      stop ' '
      end
```

The program has the following output:

```
 12.3400  .1234E+02 123.4000 1.2340E+01     .1234  .0012E+04
```

4.8.1.10 Blank Interpretation (BN, BZ)

The edit descriptors **BN** and **BZ** specify the interpretation of blanks in numeric input fields.

The **BZ** edit descriptor makes blanks, other than leading blanks, identical to zeros. Note that the blanks following the **E** or **D** in real-number input are ignored, whatever blank interpretation is in effect.

The **BN** edit descriptor "ignores" blanks; that is, it takes all the nonblank characters in the field and treats them as if they were right justified, adding as many leading blanks as there were blanks in the field.

The default, **BN**, is set at the start of each I/O statement, unless the **BLANK=** option was specified in the **OPEN** statement. If you specify a **BZ** edit descriptor, **BZ** editing is in effect until the **BN** edit descriptor is specified.

For example, look at the following program fragment:

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

If you enter any one of the following three records, terminated with an ENTER, the READ statement would interpret that record as the value 123:

```
   123
123
123   456
```

Because the repeatable edit descriptor associated with the I/O list item I is I6, only the first six characters of each record are read (three blanks followed by 123 for the first record, and 123 followed by three blanks for the last two records). Then, because blanks are ignored, all three records are interpreted as 123.

The following example shows the effect of **BN** editing with an input record that has fewer characters than the number of characters specified by the edit descriptors and *iolist*. Suppose you enter 502, followed by ENTER, in response to the following **READ** statement:

```
      READ (*,'(I5)') I
```

First, the record 502 is padded on the right with blanks to the required length, 5. If **BZ** editing were in effect, as it is by default, those two blanks would be interpreted as zeros, and the record would be equal to 50200. However, with **BN** editing in effect, the nonblank characters (502) are right justified, so the record is equal to 502.

4.8.2 Repeatable Edit Descriptors

The **I** (integer), Z (hexadecimal), **F** (single-precision real), **E** (exponent real), **G** (real with optional exponent), and **D** (double-precision real) edit descriptors are used for I/O of numeric data. The following general rules apply to all numeric edit descriptors:

- On input, leading blanks are not significant. Other blanks are interpreted differently depending on whether **BN** or **BZ** is in effect, but fields that are all blank always become the value 0. Plus signs are optional. The blanks supplied by the file system to pad a record to the required size are also not significant.

- On input with **F**, **E**, **G**, and **D** editing, an explicit decimal point appearing in the input field overrides the edit-descriptor specification of the decimal-point position.

- On output, the characters generated are right justified in the field and padded by leading blanks, if necessary.

- 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. If a real number contains more digits after the decimal point than are allowed in the field, the number is rounded.

- When reading with **I**, **Z**, **F**, **E**, **G**, **D**, or **L** edit descriptors, the input field may contain a comma that terminates the field. Reading of the next field will start at the character following the comma. The missing characters are not significant. For example, consider the following **READ** statement:

```
READ (*,'(3I5)') I,J,K
```

  Entering the following data will result in I = 1, J = 20, and K = 3:

```
1, 2 , 3,
```

  Do not use this feature if you wish to rely on explicit positional editing (that is, using the **T**, **TL**, **TR**, or *n***X** edit descriptors.)

- Two successively interpreted edit descriptors of the types **F**, **E**, **G**, and **D** are used to specify the editing of complex numbers. The types may be used in combination. The first edit descriptor specifies the real part of the complex number; the second specifies the imaginary part.

- Nonrepeatable edit descriptors may appear between repeatable edit descriptors.

The following sections describe each repeatable edit descriptor.

4.8.2.1 Integer Editing (I)

■ **Syntax**

I*w*[.*m*]

The I*w* and I*w.m* edit descriptors must be associated with an *iolist* item that is an integer. The field is *w* characters wide. On input, an optional sign may appear in the field. If the optional unsigned integer *m* is specified, input is the same as I*w*, but output is padded with leading zeros up to width *m*. For example, consider this statement:

```
WRITE(*,'(1X,I5,I5.3)')5,5
```

It prints the following on the screen:

```
 5   005
```

4.8.2.2 Hexadecimal Editing (Z)

■ **Syntax**

Z[*w*]

The Z edit descriptor is used for hexadecimal editing. During editing, internal data is processed four bits at a time. The external field is composed of hexadecimal characters (0 – 9 and A – F). Each byte of data corresponds to two hexadecimal characters (for example, 'X' corresponds to the hexadecimal characters 58).

The optional field width, *w*, specifies the number of hexadecimal characters to be read or written. If *w* is omitted, the field width defaults to 2**n*, where *n* is the length of the *iolist* item in bytes. For example, an **INTEGER*4** type is represented by eight hexadecimal characters.

On output, character data types are written in the same order in which they appear in memory. For numeric and logical types, bytes are output in order of significance, that is, from the most significant on the left to the least significant on the right. The **INTEGER*2** value 10, for example, will be output as 000A, although the order of the bytes in memory on an 8086 is actually 0A00.

The following rules of truncation and padding apply. The value *n* is the length of the *iolist* item in bytes:

Operation	Rule
Output	If $w < 2*n$, the $2*n$ hexadecimal characters are right justified and leading zeros are added to make the external field width equal to w. If $w > 2*n$, the w rightmost hexadecimal characters are output.
Input	If $w \geq 2*n$, the rightmost $2*n$ hexadecimal characters are taken from the input field. If $w < 2*n$, the w hexadecimal characters from the external field are treated as though enough leading zeros were present to make the external field width equal to $2*n$.

Blanks in an input field are treated as zeros.

To edit complex numbers, two **Z** edit descriptors must be used. The first edit descriptor specifies the real part of the complex number, the second specifies the imaginary part.

Examples

The following example demonstrates hexadecimal editing for output:

```
      CHARACTER*2 ALPHA
      INTEGER*2   INUM

      ALPHA='ZZ'
      INUM=4096

      WRITE(*,'(1X,Z,1X,Z2,1X,Z6)') ALPHA, ALPHA, ALPHA
      WRITE(*,'(1X,Z,1X,Z2,1X,Z6)') INUM, INUM, INUM
```

The above example prints the following on the screen:

```
5A5A 5A 005A5A
1000 00 001000
```

For an example of input, suppose the input record is 595A (hexadecimal characters), and the *iolist* item has CHARACTER*2 type. The record would be read as follows:

Edit Descriptor	Value Read
2	59
2X	5A
2A	59

4.8.2.3 Real Editing (F)

■ Syntax

F*w.d*

The edit descriptor **F***w.d* must be associated with an *iolist* item that is a single- or double-precision real number. 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 that 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). After these digits is an optional exponent that must be one of the following:

- **+** (plus) or **−** (minus) followed by an integer
- **E** followed by zero or more blanks, followed by an optional sign, followed by an integer

An example is the following **READ** statement:

```
      READ(*,80)xnum
80    FORMAT(F8.3)
```

The above statement reads a given input record as follows:

Input	Number Read
5	.005
-246801	-246.801
56789	5.678
-28E2	-2.800

The output field occupies w characters. One character is a decimal point, so this leaves $w-1$ characters available for digits. If the sign is negative, it must be included, leaving only $w-2$ characters available. Out of these $w-1$ or $w-2$ characters, d characters will be used for digits to the right of the decimal point. The remaining characters will be blanks or digits, as needed, in order to represent the digits to the left of 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.

■ Example

For example, look at the following program:

```
      REAL*4 g,h,e,r,k,i,n
      DATA g /12345.678/,h /12345678./,e /-4.56E+1/,r /-365./
      WRITE(*,100)g,h,e,r
100   FORMAT(1X,F8.2)
      WRITE(*,200)g,h,e,r
200   FORMAT(1X,4F10.1)
      STOP ' '
      END
```

The above program prints the following five lines on the screen:

```
12345.68
********
  -45.60
 -365.00
   12345.712345680.0      -45.6     -365.0
```

4.8.2.4 Real Editing with Exponent (E)

■ **Syntax**

E*w*.*d*[[**E***e*]]

The field is *w* characters wide. The *e* has no effect on input. The input field for the **E** edit descriptor 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 0, 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*, having one of the forms shown in Table 4.7.

Table 4.7

Forms of Exponents: E Edit Descriptor

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

The form **E***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** field. If the scale factor, *k*, is greater than $-d$ and less than or equal to 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.

4.8.2.5 Real Editing for Wide Range of Values (G)

■ Syntax

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 1, the exponent part consists of *e* digits.

G input editing is the same as **F** input editing. **G** output editing corresponds to either **E** or **F** editing, depending on the magnitude of the data. Tables 4.8 and 4.9 show how the **G** edit descriptor is interpreted.

Table 4.8

Interpretation of G Edit Descriptor

Data Magnitude	Interpretation
$m < 0.1$	**G***w*.*d* = **E***w*.*d*
$0.1 <= m < 1$	**G***w*.*d* = **F**(*w* − 4).*d*,4(' ')
$1 <= m < 10$ (i.e., $10^{(d-d)} <= m < 10^{d-(d-1)}$)	**G***w*.*d* = **F**(*w* − 4).(*d* − 1),4(' ')
$10^{(d-2)} <= m < 10^{(d-1)}$	**G***w*.*d* = **F**(*w* − 4).1,4(' ')
$10^{(d-1)} <= m < 10^{(d)}$	**G***w*.*d* = **F**(*w* − 4).0,4(' ')
$10^{(d)} <= m$	**G***w*.*d* = **E***w*.*d*

Table 4.9

Interpretation of GE Edit Descriptor

Data Magnitude	Interpretation
$m < 0.1$	**G***w*.*d***E***e* = **E***w*.*d*
$0.1 <= m < 1$	**G***w*.*d***E***e* = **F**(*w* − *e* − 2).*d*,(*e* + 2)(' ')
$1 <= m < 10$ (i.e., $10^{(d-d)} <= m < 10^{d-(d-1)}$)	**G***w*.*d***E***e* = **F**(*w* − *e* − 2).(*d* − 1),(*e* + 2)(' ')
$10^{(d-2)} <= m < 10^{(d-1)}$	**G***w*.*d***E***e* = **F**(*w* − *e* − 2).1,(*e* + 2)(' ')
$10^{(d-1)} <= m < 10^{(d)}$	**G***w*.*d***E***e* = **F**(*w* − *e* − 2).0,(*e* + 2)(' ')
$10^{(d)} <= m$	**G***w*.*d***E***e* = **E***w*.*d*

4.8.2.6 Double-Precision Real Editing (D)

- **Syntax**

D*w.d*

The I/O list item associated with a **D** edit descriptor must be a double-precision real number. All parameters and rules for the **E** edit descriptor apply to the **D** edit descriptor.

The field is w characters wide. The input field for the **D** edit descriptor 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 0, 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*, in one of the forms shown in Table 4.10.

Table 4.10

Forms of Exponents: D Edit Descriptor

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
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 form **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 **D** field. If the scale factor, k, is greater than $-d$ and less than or equal to 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.

4.8.2.7 Logical Editing (L)

■ Syntax

L*w*

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.

4.8.2.8 Character Editing (A)

■ Syntax

A[*w*]

If *w* is omitted, *w* equals 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.

When the *iolist* item is of type **INTEGER**, **REAL**, or **LOGICAL**, Hollerith data types can be used. On input, the *iolist* item becomes defined with Hollerith data; on output, the *iolist* item must be defined with Hollerith data.

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 filled in with blanks or truncated on the right to the length of *w* before being transmitted to the *iolist* item. For example, look at the following program fragment:

```
CHARACTER*10 C
READ(*,'(A15)') C
```

Assume the following thirteen characters are typed in at the keyboard:

```
ABCDEFGHIJKLM
```

The following two steps will occur:

1. The input field will be filled to fifteen characters:

   ```
   'ABCDEFGHIJKLM  '
   ```

2. The rightmost ten characters will be transmitted to the *iolist* element C:

   ```
   'FGHIJKLM  '
   ```

On output, if *w* exceeds the number of characters produced by the *iolist* item, leading blanks are provided; otherwise, the leftmost *w* characters of the *iolist* item are output.

4.8.3 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. The empty edit specification, ( ), can be used only if no items are specified in the *iolist*. When writing with an empty edit specification, a formatted **WRITE** statement writes carriage return and line feed, and a binary **WRITE** statement writes nothing. A **READ** statement with an empty edit specification skips to the next record.

If you read a record in which the total number of characters in the input record is less than the total number of characters specified by the edit descriptors and *iolist*, the following two things occur:

1. The record is padded with blanks on the right to the required length.
2. **BN** editing goes into effect. See Section 4.8.1, "Nonrepeatable Edit Descriptors," for an explanation of **BN** editing.

For example, consider the following **READ** statement:

```
READ(*,'(I5)') I
```

Assume that you enter the following as input corresponding to that READ statement:

```
 5
```

The total number of characters in the input record is two (a blank and a 5). The record is padded on the right with three blanks, but the additional blanks are ignored. The input record is thus interpreted as 5, instead of 5000.

Each item in the *iolist* is associated with a repeatable edit descriptor during the I/O statement execution. Nonrepeatable edit descriptors 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*.

During the formatted I/O process, the format controller scans and processes the format items from left to right. If the format controller encounters:

- A repeatable edit descriptor, and a corresponding item appears in the *iolist*:

  The item and the edit descriptor are associated, and I/O of that item proceeds under the format control of the edit descriptor.

- A repeatable edit descriptor, and no corresponding item appears in the *iolist*:

  The format controller terminates I/O. For the following statements, for example:

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

  the output would look like this:

```
I=       5J=
```

  The output terminates after J= because no corresponding item for the second I5 appears in the *iolist*.

- The matching final right parenthesis of the format specification, and there are no further items in the *iolist*:

  The format controller terminates I/O.

- A colon (:) edit descriptor, and there are no further items in the *iolist*:

  The format controller terminates I/O.

- A colon (:) edit descriptor, but there are further items in the *iolist*:

  The colon edit descriptor is ignored.

- The matching final right parenthesis of the format specification, and 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 on input, the remaining characters of the record are skipped. When the format controller terminates on output, an end-of-record mark is written, unless the backslash edit descriptor is used on a terminal file.

For units connected to terminal devices, the end-of-record mark is not written until the next record is written to the unit. If the device is the screen, you can use the backslash edit descriptor to suppress the end-of-record mark. For information on the backslash edit descriptor, see Section 4.8.1.7.

4.9 List-Directed I/O

A list-directed record is a sequence of values and value separators. Each value in a list-directed record must be one of the following:

- A constant, optionally multiplied by an unsigned-nonzero-integer constant. For example, `5`, or `2*5` (two successive fives) are both acceptable.
- A null value, optionally multiplied by an unsigned-nonzero-integer constant. For example, `5*` is five successive null values.

Except in string constants, none of these may have embedded blanks.

Each value separator in a list-directed record must be one of the following:

- A comma (`,`).
- A slash (`/`).

  A slash (`/`) encountered as a value separator during execution of a list-directed input statement stops execution of that statement after assignment of the previous value. Any further items in the input list are treated as if they were null values.
- One or more contiguous blanks between two constants or after the last constant.

Blanks next to value separators are ignored. For example, `5 , 6 / 7` is equivalent to `5,6/7`.

Note

List-directed I/O to or from internal files is prohibited by the ANSI standard.

4.9.1 List-Directed Input

In most cases, all the input forms available for formatted I/O are also available for list-directed formatting. This section describes all of the exceptions to this rule.

The following rules apply to list-directed input for all values:

- The form of the input value must be acceptable for the type of the input list item.
- Blanks are treated as separators and never as zeros.
- Embedded blanks can be used only within character constants, as specified in the list below.

  Note that the end-of-record mark has the effect of a blank, except when it appears within a character constant.

In addition to the rules above, the following restrictions apply to the specified values:

Type of Value	Restrictions
Single- or double-precision real 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.
Complex constants	A complex constant is an ordered pair of real or integer constants separated by a comma and surrounded by opening and closing parentheses. The first constant of the pair is the real part of the complex constant, and the second is the imaginary part.
Logical constants	A logical constant must not include either slashes or commas among the optional characters permitted for **L** editing.
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 blanks.

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 blank, comma, and slash characters.

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.

Null values

You can specify a null value in one of three ways:

1. No characters between successive value separators
2. No characters preceding the first value separator in the first record read by each execution of a list-directed input statement
3. The *r*`*` form (for example, `10*` is equivalent to 10 null values)

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.

Blanks	All blanks in a list-directed input record are considered to be part of some value separator, except for the following: • Blanks embedded in a character constant • Leading blanks in the first record read by each execution of a list-directed input statement unless immediately followed by a slash (/) or comma (,)

■ **Example**

```
C This example uses list-directed input and output
      REAL a
      INTEGER i
      COMPLEX c
      LOGICAL up,down
      DATA a/2358.2E-8/,i/91585/,c/(705.60,819.60)/
      DATA up/.TRUE./,down/.FALSE./
      OPEN(UNIT=9,FILE='listout',STATUS='NEW')
      WRITE(9,*)a,i
      WRITE(9,*)c,up,down
      REWIND(9)
      READ(9,*)a,i
      READ(9,*)c,up,down
      WRITE(*,*)a,i
      WRITE(*,*)c,up,down
      STOP ' '
      END
```

The program above has the following output:

```
  2.358200E-005         91585
 (705.6000000,819.6000000) T F
```

4.9.2 List-Directed Output

The form of the values produced by list-directed output is the same as the form of values required for input, except as noted in this section. The list-directed line size is 79 columns.

New records are created as necessary, but neither the end of a record nor blanks can occur within a constant (except in character constants). In order to provide carriage control when the record is printed, each output record begins with a blank character. Therefore, you don't have to write a blank as a carriage-control character.

In addition, the following rules apply for the specified types of data:

Type of Data	Characteristics
Logical constants	Output as **T** for the value true and **F** for the value false.
Integer constants	Output having the effect of an **I11** edit descriptor.
Single- and double-precision real constants	Output having the effect of either an **F** or **E** edit descriptor, depending on the value of the constant.
	If: — **Then:**
	$1 <= constant$ and $constant < 10^7$ — The *constant* is output using a 0PF15.6 edit descriptor for single-precision, or a 0PF24.15 edit descriptor for double-precision.
	$constant < 1$ or $constant <= 10^7$ — The *constant* is output using a 1PE15.6E2 edit descriptor for single precision, or a 1PE24.15E3 edit descriptor for double precision.
Character constants	Not delimited by apostrophes (single quotation marks). They are neither preceded nor followed by a value separator. Each internal apostrophe (single quotation mark) is represented by one externally. A blank character is inserted at the start of any record that begins with the continuation of a character constant from the preceding record.

Slashes, as value separators	Not produced by list-directed formatting.
Null values	Not produced by list-directed formatting.

■ Example

```
C This example uses list-directed output
      INTEGER i,j
      REAL a,b
      LOGICAL on,off
      CHARACTER*20 c
      DATA i/123456/,j/500/,a/28.22/,b/.0015555/
      DATA on/.TRUE./,off/.FALSE./
      DATA c/'Here''s a string'/
      WRITE(*,*)i,j
      WRITE(*,*)a,b,on,off
      WRITE(*,*)c
      STOP ' '
      END
```

The preceding program prints the following on the screen:

```
        123456           500
         28.2200000  1.555500E-003 T F
Here's a string
```

Chapter 5
Statements

5.1 Introduction

The first part of this chapter describes the kinds of statements available in Microsoft FORTRAN. The second part of the chapter contains a directory of FORTRAN statements, listed alphabetically.

A FORTRAN statement consists of an initial line, optionally followed by an unlimited number of continuation lines. Statements are written in columns 7 through 72. Statements perform actions such as computing, storing the results of computations, altering the flow of control, reading and writing files, and providing information for the compiler.

5.2 Categories of Statements

There are two basic types of statements in FORTRAN: executable and nonexecutable. An executable statement causes an action to be performed. Nonexecutable statements do not cause actions to be performed. Instead, they describe, classify, or specify the format of program elements, such as entry points, data, or program units. Table 5.1 summarizes the FORTRAN statements.

Table 5.1

Categories of FORTRAN Statements

Category	Type	Description
Assignment statements	Executable	Assign a value to a variable or an array element. See Section 5.3.1, "The ASSIGN Statement" and Section 5.3.2, "The Assignment Statement" for more information.
BLOCK DATA, **ENTRY**, **FUNCTION**, **INTERFACE**, **PROGRAM**, and **SUBROUTINE** statements	Nonexecutable	Define the start of a program unit and specify its formal arguments.
Control statements	Executable	Control the order of execution of statements. See Table 5.3.
DATA	Nonexecutable	Assigns initial values to variables.
FORMAT	Nonexecutable	Provides data-editing information.
I/O statements	Executable	Transfer data and manipulate files and records. See Table 5.4 and Chapter 4, "The Input/Output (I/O) System."
Specification statements	Nonexecutable	Define the attributes of variables, arrays, and subprograms. See Table 5.2.
Statement-function statements	Nonexecutable	Define simple, locally used functions.

Table 5.2 summarizes the specification statements.

Table 5.2
Specification Statements

Statement	Purpose
COMMON	Provides for sharing memory between two or more program units
DIMENSION	Identifies an array and defines the number of its elements
EQUIVALENCE	Specifies that two or more variables or arrays share the same memory location
EXTERNAL	Allows a user-defined subroutine or function to be passed as an argument
IMPLICIT	Defines the default type for user-defined names
INTRINSIC	Allows a predefined function to be passed as an argument
PARAMETER	Equates a constant expression with a name
SAVE	Causes variables to retain their values between invocations of the procedure in which they are defined
Type: **CHARACTER**[* *n*] **COMPLEX**[* *bytes*] **DOUBLE PRECISION** **INTEGER**[* *bytes*] **LOGICAL**[* *bytes*] **REAL**[* *bytes*]	Specifies the type of user-defined names

Table 5.3 summarizes the control statements.

Table 5.3
Control Statements

Statement	Purpose
CALL	Calls and executes a subroutine
CONTINUE	Does not have any effect; often used as target of **GOTO**, or as the terminal statement in a **DO** loop
DO	Causes repetitive evaluation of the statements in the **DO** loop, 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 and optionally executes operating-system commands
RETURN	Returns control to the program unit that called a subroutine or function
STOP	Terminates a program

Table 5.4 summarizes the I/O statements.

Table 5.4
I/O Statements

Statement	Purpose
BACKSPACE	Positions the file connected to the specified unit to the beginning of the previous record
CLOSE	Disconnects the specified unit 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 file or unit
LOCKING	Locks direct-access files and records
OPEN	Associates a unit number with an external device or with a file
PRINT	Specifies output to the screen
READ	Transfers data from a file to the items in an I/O list
REWIND	Repositions a specified unit to the first record in the associated file
WRITE	Transfers data from the items in an I/O list to a file

5.3 Statement Directory

The rest of this chapter is an alphabetical listing of all Microsoft FORTRAN statements. Each statement is described using the following format:

Heading	Information
■ **Action**	Summary of what the statement does.
■ **Syntax**	Correct syntax for the statement, and description of the statement's parameters.
■ **Remarks**	Use of the statement.
■ **Example**	Sample programs or program fragments that illustrate the use of the statement. This section does not appear with every reference entry.

Note

The syntax of statements that do not fit on one line is shown on more than one line, as in the following example:

CLOSE ([**UNIT=**]*unitspec*
[,**ERR=***errlabel*]
[,**IOSTAT=***iocheck*]
[,**STATUS=***status*])

When you use these statements, you must still follow the formatting rules described in Section 3.2, "Lines," or, if the $FREEFORM metacommand is specified, the rules given in Section 3.4, "Free-Form Source Code." The following program fragment, for example, is illegal:

```
CLOSE (UNIT=2,
ERR=100,
IOSTAT=errvar)
```

Either of the following two statements, however, is correct:

```
CLOSE (UNIT=2,ERR=100,IOSTAT=errvar)

CLOSE (UNIT=2,
+ERR=100,
+IOSTAT=errvar)
```

5.3.1 The ASSIGN Statement (Label Assignment)

■ Action

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

■ Syntax

ASSIGN *label* **TO** *variable*

Parameter	Description
label	Format label or statement label. The *label* referred to must appear in the same program unit as the **ASSIGN** statement.
variable	An integer variable.

■ Remarks

Use the **ASSIGN** statement to assign the value of a label to a variable. You can use variables with label values in the following situations:

Situation	Use
An assigned **GOTO** statement	The assigned **GOTO** statement requires a variable that must have the value of the label of an executable statement.
A format specifier	Input/output statements allow you to use a variable to specify the label of a **FORMAT** statement.

The value of a label is not the same as the label number; the label is instead identified by a number assigned by the compiler. For example, the value of IVBL in the following is not 400:

```
ASSIGN 400 TO IVBL
```

Therefore, variables used in **ASSIGN** statements are not defined as integers. If you want to use a variable defined by an **ASSIGN** statement in an arithmetic expression, you must first define the variable by a computational assignment statement (see Section 5.3.2) or by a **READ** statement.

If you use **INTEGER*1** variables for *variable*, note that **INTEGER*1** variables can only be used for the first 128 **ASSIGN** statements in a subprogram.

■ Example

```
C Assign statement label 100 to the integer variable
C ivar
      ASSIGN 100 TO ivar
C Use ivar as a FORMAT statement label
      WRITE(*,ivar)
C Assign statement label 200 to ivar
      ASSIGN 200 TO ivar
C Use ivar as the target label of an assigned GOTO
C statement
      GOTO ivar
      WRITE (*,*)' This is never written'
  200 CONTINUE
      WRITE (*,*)' This is written'
  100 FORMAT (' This is format 100')
      END
```

5.3.2 The Assignment Statement (Computational)

■ Action

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

■ Syntax

variable = *expression*

Parameter	Description
variable	A variable or array-element reference
expression	Any expression

■ Remarks

The type of the variable or array element and the type of expression must be compatible, as follows:

- If *expression* is numeric, then *variable* must be numeric, and the statement is called an arithmetic assignment statement. If the data types of *expression* and *variable* are not identical, *expression* is converted to the data type of *variable*.

  Section 2.7.1.2, "Type Conversion of Arithmetic Operands," explains how integer, real, and complex numbers are converted.

- If *expression* is logical, then *variable* must be logical, and the statement is called a logical assignment statement.

  Logical *expressions* of any size can be assigned to logical *variables* of any size without affecting the value of *expression*.

  Note that integer and real *expressions* may not be assigned to logical *variables*, nor may logical *expressions* be assigned to integer or real *variables*.

- If *expression* has the type **CHARACTER**, the statement is called a character assignment statement. If the **$NOTSTRICT** metacommand (the default) is in effect, then a character expression can be assigned to a noncharacter variable, and a noncharacter variable or array element (but not an expression) can be assigned to a character variable. If **$STRICT** is in effect, both *variable* and *expression* must have type **CHARACTER**.

  For character assignment statements, if the length of *expression* does not match the size of *variable*, *expression* is adjusted as follows:

  - If *variable* is longer than *expression*, then *expression* is padded with blanks on the right.
  - If *variable* is shorter than *expression*, then characters on the right of *expression* are ignored.

■ Examples

The following program demonstrates assignment statements:

```
      REAL              A, B, C
      LOGICAL           ABIGGER
      CHARACTER*5       ASSERTION

      C = .01
      A = SQRT(C)
      B = C**2

      ASSERTION = 'A > B'
      ABIGGER   = (A .GT. B)

      WRITE (*,100) A, B
100   FORMAT(' A =', F7.4, '    B =', F7.4)

      IF (ABIGGER) THEN
            WRITE(*,*) ASSERTION, ' is true.'
         ELSE
            WRITE(*,*) ASSERTION, ' is false.'
      ENDIF
      END
```

The program above has the following output:

```
A =   .1000     B =   .0001
A > B is true.
```

The following program fragment demonstrates legal and illegal assignment statements:

```
        INT i
        REAL x
        CHAR c1
        x=2.0
C The following 2 statements are legal:
        c1=x
        i=c1
C The following 2 statements are illegal:
        c1=x+1.0
        i=c1//'test'
```

5.3.3 The BACKSPACE Statement

■ Action

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

■ Syntax

BACKSPACE {*unitspec* |
(**[UNIT=]***unitspec*
[,ERR=*errlabel***]**
[,IOSTAT=*iocheck***]**)}

Parameter	Description
unitspec	An external unit specifier (see Section 4.3.2 for information). If *unitspec* has not been opened, a run-time error is produced.
errlabel	The label of an executable statement in the same program unit as this **BACKSPACE** statement. If *errlabel* is specified, I/O errors transfer control to the statement at *errlabel*. If *errlabel* is omitted, I/O errors cause run-time errors. The effects of I/O errors are determined by the presence of *iocheck*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."
iocheck	An integer variable or integer array element that becomes defined as 0 if no error is encountered, or as a positive integer if an error is encountered. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

■ Remarks

The **BACKSPACE** statement backs up exactly one record, except in the special cases listed below:

If:	Then:
There is no preceding record	The file position is not changed.
The preceding record is the end-of-file record	The file is positioned before the end-of-file record.
The file position is in the middle of the record	The file is positioned to the start of that record.

If a parameter of the **BACKSPACE** statement is an expression that calls a function, that function must not cause an I/O statement or the EOF intrinsic function to be executed, because the results are unpredictable.

■ Examples

```
C EXAMPLES OF BACKSPACE STATEMENTS
      BACKSPACE 5
      BACKSPACE LUNIT
      BACKSPACE (5)
      BACKSPACE (UNIT = LUNIT, ERR = 30, IOSTAT = IOS)
```

5.3.4 The BLOCK DATA Statement

■ Action

Identifies a block-data subprogram, where variables and array elements in named common blocks can be initialized

■ Syntax

BLOCK DATA [*blockdataname*]

Parameter	Description
blockdataname	An optional global symbolic name for the subprogram identified by the **BLOCK DATA** statement. This name must not be the same as any of the names for local variables or array elements defined in the subprogram labeled by *blockdataname*, and must not be the same as any of the names given to the main program, external procedures, common blocks, or other block-data subprograms.

■ Remarks

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

Only one unnamed block-data subprogram may appear in the executable program. Otherwise, the default name will be defined twice, generating an error.

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

- The only statements that may be used in a block-data subprogram are **BLOCK DATA**, **COMMON**, **DIMENSION**, **PARAMETER**, **IMPLICIT**, **EQUIVALENCE**, **SAVE**, **DATA**, **END**, and type statements. No executable statements are permitted.

- Named common blocks specified in block-data subprograms must have unique names. Only an entity defined in a named common block may be initially defined in a block-data subprogram.
- All the constituents of a named common block specified in a block-data subprogram must be specified in that block-data subprogram, even if not all the constituents are initialized, as shown in the following examples. This is because the length of the named common block cannot change between subprograms.

■ Examples

```
C The following block-data subprogram initializes
C one constituent of the named common block /LAKES/:
C
      BLOCK DATA total
      COMMON /LAKES/ Wawenpaupak, Hopatcong, Great(5)
      DATA Hopatcong /78/
      END
C
C Now, assuming the use of the same common block,
C /LAKES/, the following block-data subprogram is
C NOT allowed. This is because not all the constituents
C of /LAKES/ are specified.
C
      BLOCK DATA total
      COMMON /LAKES/ Hopatcong
      DATA Hopatcong /78/
      END
```

5.3.5 The CALL Statement

■ Action

Calls and executes a subroutine from another program unit

■ Syntax

CALL *subr*[([*actuals*])]

Parameter	Description
subr	The name of the subroutine to be called.
actuals	One or more optional actual arguments. If there is more than one actual argument, they must be separated by commas. Each actual argument can be an alternate-return specifier (**label*); a constant, variable, or expression; an array or array element; the name of a subroutine or external function; the name of an intrinsic function that can be passed as an argument; or a Hollerith constant.

■ Remarks

Execution of a **CALL** statement proceeds as follows:

1. Arguments that are expressions are evaluated.
2. Actual arguments are associated with their corresponding formal arguments.
3. The body of the specified subroutine is executed.
4. Control is returned to the calling routine, either to a statement specified by an alternate return or to the statement following the **CALL** statement.

A subroutine can be called from any program unit.

Microsoft FORTRAN does not support recursive subroutine calls. Moreover, it cannot detect them. That is, a subroutine cannot call itself directly, nor can it call another subroutine that results in the first subroutine's being called again before the first subroutine returns control to its caller. Recursive subroutine calls do not produce an error message. Programs that contain recursive calls have undefined results.

There must be the same number of actual arguments in the **CALL** statement as there are formal arguments in the corresponding **SUBROUTINE** statement, unless the C and VARYING attributes (described in Sections 2.6.2 and 2.6.10) have been used to declare the subroutine. Formal arguments and their corresponding actual arguments must have the same data type except in the case of Hollerith constants. When the actual argument is a Hollerith constant, the formal argument need not be the same type, as long as it is of type INTEGER, REAL, or LOGICAL. If a **SUBROUTINE** statement lacks formal arguments, a **CALL** statement referencing that subroutine must not have any actual arguments. However, a pair of empty parentheses can follow *subr*.

For all arguments passed by reference (see Section 2.6.9, "VALUE," for information on passing arguments by value), the compiler assumes that the type of the formal argument is the same as the type of the corresponding actual argument. If the type of the formal argument is known, it is used only to check that the arguments have the same data type.

The compiler checks that the actual arguments used in different calls to the same subroutine correspond in number and type. If the **SUBROUTINE** statement, or an INTERFACE statement that defines the subroutine, is in the same source file as any calls to it, the compiler ensures that all actual arguments agree with the subroutine's formal arguments.

If arguments do not agree, and they have not been checked as described above, your program will have unpredictable results.

Note

When passing integer and logical arguments, be especially careful about argument agreement. The setting of the **SSTORAGE** metacommand affects how integer and logical arguments are passed. When the default **SSTORAGE:4** is in effect, all actual arguments that are integer or logical constants or expressions are assigned to **INTEGER*4** or **LOGICAL*4** temporary variables. When **SSTORAGE:2** is in effect, all actual arguments that are integer or logical constants or expressions are assigned to temporary variables of type **INTEGER*2** or **LOGICAL*2**. To pass 2-byte integer arguments when **SSTORAGE:4** is in effect, or 4-byte integer arguments when **SSTORAGE:2** is in effect, use the **INT2** and **INT4** intrinsic functions, as described in Section 3.11.3.1.

The alternate-return feature lets you specify the statement to which a subroutine should return control. To use the alternate-return feature, do the following:

1. Choose the statements in the calling routine to which you wish to return control. Enter the labels of these statements, preceded by asterisks, in the actual argument list of the **CALL** statement, as in this statement:

```
CALL INVERT (*100, *200, row, *500, column)
```

2. In the corresponding **SUBROUTINE** statement, enter asterisks for the formal arguments corresponding to the ***label** actual arguments in the **CALL** statement, as in the following **SUBROUTINE** statement:

```
SUBROUTINE INVERT (*, *, first, *, second)
```

3. In the subroutine, have at least one **RETURN** statement for each alternate return. As arguments for these **RETURN** statements, specify a 1 for the **RETURN** statement that should return control to the first statement label in the **CALL** statement, a 2 for the **RETURN** statement that should return control to the second statement label in the **CALL** statement, and so on.

eclarator. Specifying
n array. See Section
N Statement," for a
declarators.

stant in the range 1
er constant expression
between 1 and
an asterisk in
th parameter speci-
of the *vname* immedi-
alue, if specified,
ated by *bytes*.

plicit type of
n unit and cannot
nit.

ength is specified
ollowing cases:

atements

ction and defined

to 1.

tements.

t RETURN 1 is reached in the program
ts in steps 1 and 2 above, control
abel 100 in the calling routine. If
ol returns to the statement at label 200;
l, control returns to the statement at
tement without any number is reached
statement that has a number for which
turn label is reached (such as RETURN
returns to the statement following the
ling routine.

```
NT
LL ERROR(IERR)


.RRNO)
:NO
. I5, 'DETECTED')


es the alternate return feature:
nt,*10,J,*20,*30)
l return'

ned to 10'

ned to 20'

ned to 30'
```

```
ANG (I,*,J,*,*)
RN 1
RN 2
RN 3
```

5.3.6 The CHARACTER Statement

■ Action

Specifies that user-defined names are of the character data type

■ Syntax

CHARACTER[*bytes] *vname* [[(*dim*)] [**length*]] [/ *values* /]
[,*vname* [[(*dim*)] [**length*]] [/ *values* /] ...

The order of the *dim* and *length* parameters can be reversed.

Parameter	Description
bytes	An optional unsigned integer constant in the range 1 through 32,767, an integer constant expression (evaluating to an integer between 1 and 32,767) in parentheses, or an asterisk in parentheses (*). The *bytes* parameter specifies the length, in bytes, of the items specified in the **CHARACTER** statement. This value can be overridden by the *length* parameter.
vname	The symbolic name of a constant, variable, array, external function, statement function, or intrinsic function; or a function subprogram or an array declarator. See Section 5.3.12, "The DIMENSION Statement," for information on array declarators. The parameter *vname* cannot be the name of a subroutine or main program.
attrs	An optional list of attributes, separated by commas. The *attrs* describe *vname*. The following attributes can be used with *vname*: ALIAS, C, EXTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE. See Section 2.6, "Attributes," for information on attributes.

dim — An optional dimension declarator. Specifying *dim* declares *vname* as an array. See Section 5.3.12, "The DIMENSION Statement," for a description of dimension declarators.

length — An unsigned integer constant in the range 1 through 32,767, an integer constant expression (evaluating to an integer between 1 and 32,767) in parentheses, or an asterisk in parentheses (*). The *length* parameter specifies the length, in bytes, of the *vname* immediately preceding it. This value, if specified, overrides the length indicated by *bytes*.

values — A list of constants and repeated constants, separated by commas. A repeated constant is written in the form *n***constant*, where *n* is a positive nonzero integer constant, and is equivalent to *constant* repeated *n* times. The */values/* option, if specified, initializes *vname*. The following statement, for example, declares that word is of type character and sets word equal to 'start':

```
CHARACTER*5 word /'start'/
```

■ Remarks

A **CHARACTER** statement confirms or overrides the implicit type of *vname*. The name *vname* is defined for the entire program unit and cannot be defined by any other type statement in that program unit.

An asterisk ((*)) as a length specifier indicates that the length is specified elsewhere. An asterisk length specifier is allowed in the following cases:

- Character constants defined by **PARAMETER** statements
- Formal character arguments
- Character functions that are referenced in one function and defined with a specific length in the same program unit

If neither *length* nor *bytes* is specified, the length defaults to 1.

CHARACTER statements must precede all executable statements.

■ Examples

```
C EXAMPLES OF CHARACTER STATEMENTS
      CHARACTER WT*10, CITY*80, CH
      CHARACTER name(10)*20
```

5.3.7 The CLOSE Statement

■ Action

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

■ Syntax

CLOSE ([**UNIT=**]*unitspec*
[,**ERR=***errlabel*]
[,**IOSTAT=***iocheck*]
[,**STATUS=***status*])

Parameter	Description
unitspec	An external unit specifier. If the optional string **UNIT=** is not specified, *unitspec* must be the first parameter. See Section 4.3.2 for information about unit specifiers. The unit specified by *unitspec* does not have to exist.
errlabel	The label of an executable statement in the same program unit as this statement. If *errlabel* is specified, I/O errors transfer control to the statement at *errlabel*. If *errlabel* is omitted, I/O errors cause run-time errors. The effects of I/O errors are determined by the presence of *iocheck*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."
iocheck	An integer variable or integer array element that becomes defined as zero if no error is encountered, or as a positive integer if an error is encountered. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

status — A character expression that evaluates to either **'KEEP'** or **'DELETE'** when trailing blanks are removed.

For files opened as scratch files, the default for *status* is **'DELETE'**. Scratch files are always deleted upon normal program termination; specifying **STATUS='KEEP'** for scratch or temporary files produces a run-time error. The default for *status* for all other files is **'KEEP'**.

■ Remarks

Opened files do not have to be explicitly closed. Normal termination of a Microsoft FORTRAN program will close each file with its default status. Closing of unit zero automatically reconnects unit zero to the keyboard and screen. Closing of units five and six automatically reconnects those units to the keyboard or screen, respectively. Closing of the asterisk (*) unit results in a compile-time error.

If a parameter of the **CLOSE** statement is an expression that calls a function, that function must not cause an I/O statement or the END= specifier statement to be executed, because the results are unpredictable.

■ Example

```
C CLOSE AND DISCARD FILE
      CLOSE(7,STATUS='DELETE')
```

5.3.8 The COMMON Statement

■ Action

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

■ Syntax

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

Parameter	Description
cname	An optional common-block name. The default for *cname* is the blank common block.
attrs	A list of attributes, separated by commas. The attributes describe *cname*. Only **ALIAS**, **C**, **FAR**, **NEAR**, and **PASCAL** can be used with common-block names. See Section 2.6, "Attributes," for more information.
nlist	A mandatory list of variable names, array names, and array declarators, separated by commas. See Section 5.3.12, "The DIMENSION Statement," for information on array declarators. 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.

■ Remarks

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

An item that appears in *nlist* cannot be initialized in a type statement. The following example causes an error:

```
INTEGER I
COMMON I
DATA I /1/
```

In the $STRICT metacommand is specified, all items in a common block must be either character or noncharacter items. When $NOTSTRICT, the default, is in effect, Microsoft FORTRAN allows the mixture of character and noncharacter variables and arrays in common blocks. Microsoft FORTRAN restricts noncharacter variables to even-byte addresses, so the association of character and noncharacter variables within a common block can be affected. Because of the order requirement, the compiler cannot adjust the position of variables within a common block to comply with the even-address restriction. The compiler generates an error message for those associations which result in a conflict. Placing all character variables at the end of a common block can make it easier to avoid these conflicts.

The length of a common block equals 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.

■ 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 IP, JX, Z, IDUM(10)
      COMMON /MYCOM/ A(3)
      .
      .
      END
```

5.3.9 The COMPLEX Statement

■ Action

Specifies that user-defined names are of type complex

■ Syntax

COMPLEX[*bytes*] *vname*[(*attrs*)][* *length*][(*dim*)][/ *values* /]
[,*vname*[(*attrs*)][* *length*][(*dim*)][/ *values* /]]...

The order of the *length* and *dim* parameters can be reversed.

Parameter	Description
bytes	Must be 8 or 16. The *bytes* parameter specifies the length, in bytes, of the items specified by the **COMPLEX** statement. This value can be overridden by the *length* parameter.
vname	The mandatory symbolic name of a constant, variable, array, external function, statement function, or intrinsic function; or a function subprogram or an array declarator. Cannot be the name of a subroutine or a main program.
attrs	An optional list of attributes separated by commas. The *attrs* describe *vname*. The following attributes can be used with *vname*: **ALIAS**, **C**, **EXTERN**, **FAR**, **HUGE**, **NEAR**, **PASCAL**, **REFERENCE**, **VALUE**.
length	Assigns *length* to *vname*. The value of the *length* parameter must be 8 or 16. If *length* is specified, it overrides the length attribute specified by *bytes*.
dim	A dimension declarator. You can only specify *dim* if *vname* is an array. If *dim* is specified, the **COMPLEX** statement declares the array *vname*.

values A list of constants and repeated constants, separated by commas. A repeated constant is written in the form *n***constant*, where *n* is a positive-nonzero-integer constant, and is equivalent to the constant *constant* repeated *n* times. The */values/* option, if specified, initializes *vname*. The following statement, for example, declares that num is of type complex, and sets num equal to (32,0,0,0):

```
COMPLEX num /(32,0,0,0)/
```

■ Remarks

A **COMPLEX** statement confirms or overrides the implicit type of *vname*. The name *vname* is defined for the entire program unit and cannot be defined by any other type statement in that program unit.

COMPLEX statements must precede all executable statements.

■ Examples

```
C EXAMPLES OF COMPLEX STATEMENTS
      COMPLEX NAME, CH ZDIF*8, XDIF*16
      COMPLEX*8 ZZ
      COMPLEX*16 ax,by
      COMPLEX x*16,y(10)*8,z*16(10)
```

5.3.10 The CONTINUE Statement

■ Action

Does not have any effect

■ Syntax

CONTINUE

■ Remarks

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

■ Example

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

5.3.11 The DATA Statement

■ Action

Assigns initial values to variables

■ Syntax

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

Parameter	Description
nlist	A list of variables, array elements, array names, substring names, and implied-**DO** lists, separated by commas. Implied-**DO** lists are discussed in the "Remarks" section below. Each subscript expression in *nlist* must be an integer constant expression, except for implied-**DO** variables. Each substring expression in *nlist* must be an integer constant expression.
clist	A list of constants and/or repeated constants and/or Hollerith constants, separated by commas. A repeated constant is written in the form *n***c*, where the repeat factor *n* is a positive-nonzero-integer constant, and *c* is the constant to be repeated. The repeated constant `5*10`, for example, is equivalent to the *clist* `10,10,10,10,10`. A repeat factor followed by a constant is equivalent to a list of all constants having the specified value and repeated as often as specified by the repeat constant. A Hollerith constant is written in the form *n***H***data*, where *n* is a positive-nonzero-integer constant, and *data* is a string of *n* characters.

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 column-major order. Array elements can be indexed only by constant subscripts.

■ Remarks

Type conversion takes place for each noncharacter element in *clist*.

If the $STRICT metacommand is not specified, a character element in a *clist* can correspond to a variable of any type. If the length of the character element is less than the length of that variable or array element, it is extended to the length of the variable by adding blank characters at the right. If the character element is longer than the variable or array element, it is truncated. A single character constant defines one variable or one array element. A repeat count can be used.

Only local variables and array elements can appear in a **DATA** statement (unless the **DATA** statement is in a block-data subprogram). Formal arguments, variables in blank common blocks, and function names cannot be assigned initial values with a **DATA** statement. Elements in named common blocks can be assigned initial values by using a **DATA** statement in a block-data subprogram.

The form of an implied-**DO** list is as follows:

(*dlist*,*dovar*=*start*,*stop* [[,*inc*]])

Parameter	Description
dlist	A list of array-element names and implied-**DO** lists.
dovar	The name of an integer variable, called the implied-**DO** variable.
start, *stop*, and *inc*	Integer constant expressions. Each expression can contain implied-**DO** variables (*dovar*) of other implied-**DO** lists that have this implied-**DO** list within their ranges.

For example, the following are implied-**DO** lists:

```
(count(i), i=5,15,2)
((array(sub,low), low=1,12),sub=1,2)
((result(first,second),first=1,max),second=1,upper)
```

An iteration count and the values of the implied-**DO** variable are established from *start*, *stop*, and *inc* exactly as for a **DO** loop except that the iteration count must be positive. See Section 5.3.13, "The DO Statement," for more information. When an implied-**DO** list appears in a **DATA** statement, the list items in *dlist* are initialized once for each iteration of the implied-**DO** list. The range of an implied-**DO** list is *dlist*. If the program contains another variable with the same name as *dovar*, that variable is not affected by the use of *dovar* in a **DATA** statement.

■ Example

```
INTEGER N, ORDER, ALPHA, list(100)
REAL COEF(4), EPS(2), pi(5), x(5,5)
CHARACTER*12 help
DATA N /0/, ORDER /3/
DATA ALPHA /'A'/
DATA COEF /1.0,2*3.0,1.0/, EPS(1) /.00001/
DATA ((X(J,I), I=1,J), J=1,5) /15*0. /
DATA pi /5*3.14159/
DATA list /100*0/
DATA help(1:4), help(5:8), help (9:12) /3*'HELP'/
```

5.3.12 The DIMENSION Statement

■ Action

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

■ Syntax

DIMENSION *array* [[*attrs*]] ([[*lower*:]*upper*) [[,*array* [[*attrs*]] ([[*lower*:]*upper*)]]...

Parameter	Description
array	The name of an array.
attrs	A list of attributes separated by commas. The *attrs* describe *array*. The following attributes can be used with *array*: **ALIAS**, **C**, **EXTERN**, **FAR**, **HUGE**, **NEAR**, **PASCAL**, **REFERENCE**, **VALUE**.
lower	The lower dimension bound, which can be positive, negative, or zero. The default for *lower* is one.
upper	The upper dimension bound, which can be positive, negative, zero, or an asterisk. It must be greater than or equal to *lower*.

The specifier [[*lower*:]*upper* is also called a "dimension declarator." The number of dimensions in the array equals the number of dimension declarators specified. There is no limit on the number of dimensions. The specifier *array*([[*lower*:]*upper*) is also called an "array declarator."

It can be useful to specify both the upper and lower dimension bounds. If, for example, one array contains data from experiments numbered 28 through 112, you could dimension the array as follows:

```
DIMENSION exprmt (28:112)
```

Then, to refer to the data from experiment 72, you would reference `exprmt(72)`.

You can use any of the following as dimension bounds:

Bound	Description
An arithmetic constant	If all of an array's dimensions are specified by arithmetic constants, the array has a constant size. The arithmetic value is truncated to an integer.
A nonarray-integer formal argument or a nonarray-integer variable in a common block in the same program unit as the **DIMENSION** statement	The dimension is defined as the initial value of the variable upon entry to the subprogram at execution time. If a dimension bound of *array* is an integer formal argument or an integer variable in a common block, the array is an "adjustable-size array." The variable must be given a value before the subprogram containing the adjustable-size array is called.
An asterisk	Only *upper* can be an asterisk, and an asterisk can only be used for *upper* in the last dimension of *array*. If *upper* is an asterisk, then *array* is an "assumed-size array." For an assumed-size array, the subprogram array is defined at execution time to be the same size as the array in the calling program. The following **DIMENSION** statement defines an assumed-size array in a subprogram: `DIMENSION data (19,*)` At execution time, the array `data` is given the size of the corresponding array in the calling program.
An arithmetic expression	Expressions cannot contain references to functions or array elements. Expressions can contain variables only in adjustable-size arrays. The result of the expression is truncated to an integer.

■ Remarks

All adjustable- and assumed-size arrays, as well as the bounds for adjustable-size arrays, must also be formal arguments to the program unit in which they appear.

Array elements are stored in column-major order: the leftmost subscript is incremented first when the array is mapped into contiguous memory addresses.

For example, look at the following statements:

```
INTEGER*2 A (2, 0:2)
DATA A /1, 2, 3, 4, 5, 6/
```

If A is placed at location 1000 in memory, these statements cause the following mapping:

Array Element	Address	Value
A (1,0)	1000	1
A (2,0)	1002	2
A (1,1)	1004	3
A (2,1)	1006	4
A (1,2)	1008	5
A (2,2)	100A	6

■ Examples

The following program dimensions two arrays:

```
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
```

The following program uses assumed- and adjustable-size arrays:

```
      real        magnitude, minimum
      integer     vecs, space, vec

C The array vecs holds 4 three-dimensional vectors.
      dimension   vecs(3,4)
      data        vecs/1,1,1, 2,1,0, 3,4,7, -2,2,1/

C Find minimum magnitude.
      minimum=1E10
      do 100 vec = 1,4

C Call the function magnitude to calculate the magnitude of
C vector vec.
         minimum = AMIN1(minimum, magnitude(vecs, 3, vec))
 100  continue

      write(*,110) minimum
 110  format(' Vector closest to origin has a magnitude of',
     1           f12.6)
      end


C Function returns the magnitude of the j-th column vec in a
C matrix. Note that, because of the assumed-size array, the
C subroutine does not need to know the number of columns in
C the matrix. It only requires that the specified column
C vector be a valid column in the matrix. The number of rows
C must be passed so the function can do the sum.

      real function MAGNITUDE(matrix, rows, j)

            real        sum
            integer     matrix, rows, i, j
            dimension   matrix (rows,*)

            sum = 0.0
            do 100 i = 1, rows
               sum = sum + matrix(i,j)**2
 100        continue
            MAGNITUDE = SQRT(sum)
            return
      end
```

5.3.13 The DO Statement

■ Action

Repeatedly evaluates the statements following the **DO** statement through the statement at *label*

■ Syntax

DO *label* [[,]] *dovar*=*start*,*stop* [[,*inc*]]

Parameter	Description
label	The mandatory statement label of an executable statement.
dovar	A mandatory integer, real, or double-precision variable, called the "**DO** variable."
start, *stop*	Mandatory integer, real, or double-precision expressions.
inc	An optional integer, real, or double-precision expression. The parameter *inc* cannot be equal to zero; *inc* defaults to one.

■ Remarks

The *label* must appear after the **DO** statement and be contained in the same program unit. The statement at *label* 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** statement, it may contain any executable statement permitted within a logical **IF** statement.

The range of a **DO** loop begins with the statement immediately following the **DO** statement and includes the terminal statement of the **DO** loop.

Execution of a **CALL** statement that is in the range of a **DO** loop does not cause the **DO** loop to become inactive, unless an alternate-return specifier in a **CALL** statement returns control to a statement outside the **DO** loop.

The following restrictions affect the execution of a **DO** statement:

- 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.
- If a **DO** statement appears within an **IF**, **ELSEIF**, or **ELSE** block, the range of the **DO** loop must be entirely contained in the block.
- 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 *dovar* 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.

Note that the number of iterations in a **DO** loop is limited to the precision of the arithmetic used to evaluate it. For example, **DO** loops that use **INTEGER*2** variables as the **DO** variable and bounds cannot be evaluated more than 32,767 times. If the **$DEBUG** metacommand is set, an error is generated if a **DO** variable overflows. If **$DEBUG** is not set, the results are unpredictable. In the following program fragment, for example, the number of iterations is 64,001, and the **DO** variable I can only be evaluated 32,767 times:

```
$DEBUG
C This causes an error:
      INTEGER*2 I
      DO 10 I = -32000,32000,1
```

When a **DO** statement is executed, the following steps occur:

1. The expressions *start*, *stop*, and *inc* are evaluated. If necessary, type conversion is performed. The **DO** variable *dovar* is set to the value of *start*.

2. The iteration count for the loop is tested.

   The iteration count for the loop is the following:

   MAX(INT((*stop* − *start* + *inc***) / ***inc***),0)**

If either of the following is true, the iteration count can be zero:

- *start* > *stop* and *inc* > 0
- *start* < *stop* and *inc* < 0

3. If the iteration count is greater than zero, the statements in the range of the **DO** loop are executed.

   Note that if the **SDO66** metacommand is in effect, the statements in the range of the **DO** loop are executed at least once.

4. After the execution of the terminal statement of the **DO** loop, the value of the **DO** variable *dovar* is increased by the value of *inc* that was computed when the **DO** statement was executed.

5. The iteration count is decreased by one.

6. The iteration count is tested. If the iteration count is greater than 0, the statements in the range of the **DO** loop are executed again.

■ Examples

The following two program fragments are examples 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
```

5.3.14 The DOUBLE PRECISION Statement

■ Action

Specifies the **DOUBLE PRECISION** type for user-defined names

■ Syntax

DOUBLE PRECISION *vname*[[*attrs*]][(*dim*)][/*values*/]
[[,*vname*[[*attrs*]][(*dim*)][/*values*/]]...

Parameter	Description
vname	The mandatory symbolic name of a constant, variable, array, external function, statement function, or intrinsic function; or the name of a function subprogram or an array declarator. Cannot be the name of a subroutine or main program.
attrs	An optional list of attributes separated by commas. The *attrs* describe *vname*. The following attributes can be used with *vname*: ALIAS, C, EXTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE.
dim	A dimension declarator. If *dim* is specified, the **DOUBLE PRECISION** statement declares the array *vname*.
values	A list of constants and repeated constants, separated by commas. A repeated constant is written in the form *n***constant*, where *n* is a positive-nonzero-integer constant, and is equivalent to *constant* repeated *n* times. The /*values*/ option, if specified, initializes *vname*. The following statement, for example, declares that num is of type DOUBLE PRECISION and sets num equal to 56.12:

```
DOUBLE PRECISION num /56.12D0/
```

■ Remarks

A **DOUBLE PRECISION** statement confirms or overrides the implicit type of *vname*. The name *vname* is defined for the entire program unit, and cannot be defined by any other type statement in that program unit.

DOUBLE PRECISION statements must precede all executable statements in a program unit.

■ Example

```
C EXAMPLE OF DOUBLE PRECISION STATEMENT
      DOUBLE PRECISION X
```

5.3.15 The ELSE Statement

■ Action

Marks the beginning of an **ELSE** block

■ Syntax

ELSE

■ Remarks

An **ELSE** block consists of any executable statements between the **ELSE** statement and the next **ENDIF** statement at the same **IF** level as this **ELSE** statement. The matching **ENDIF** statement must appear before any **ELSE** or **ELSEIF** statements of the same **IF** level. See Section 5.3.30, "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
.
.
```

5.3.16 The ELSEIF Statement

■ Action

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

■ Syntax

ELSEIF (*expression*) **THEN**

Parameter	Description
expression	A logical expression

■ Remarks

The associated **ELSEIF** block consists of any executable statements between the **ELSEIF** statement and the next **ELSEIF**, **ELSE**, or **ENDIF** statement at the same **IF** level.

When the **ELSEIF** statement is executed, *expression* is evaluated, and the following steps are performed:

If:	Then:
The *expression* is true, and there is at least one executable statement in the **ELSEIF** block	The next statement executed is the first statement of the **ELSEIF** block.
The *expression* is true, and there are no executable statements in the **ELSEIF** block	The next statement executed is the next **ENDIF** statement at the same **IF** level as the **ELSEIF** statement.
The *expression* is 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 5.3.30, "The IF THEN ELSE Statement," for a discussion of **IF** levels.

After the last statement in the **ELSEIF** block has been executed, 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
```

5.3.17 The END Statement

■ Action

Terminates execution of the main program. In a subprogram, the **END** statement has the same effect as a **RETURN** statement. The **END** statement always marks the end of the program unit in which it appears.

■ Syntax

END

■ Remarks

The **END** statement must appear as the last statement in every program unit. An **END** statement must appear alone on an initial line (that is, not a continuation line), with no label. No continuation lines may follow an **END** statement. No other FORTRAN 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
```

5.3.18 The ENDFILE Statement

■ Action

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

■ Syntax

ENDFILE {*unitspec* |
(⟦**UNIT=**⟧*unitspec*
⟦,**ERR=***errlabel*⟧
⟦,**IOSTAT=***iocheck*⟧)}

Parameter	Description
unitspec	An external unit specifier (see Section 4.3.2 for information). If *unitspec* has not been opened, a run-time error is produced.
errlabel	The label of an executable statement in the same program unit as the current **ENDFILE** statement. If *errlabel* is specified, I/O errors transfer control to the statement at *errlabel*. If *errlabel* is omitted, I/O errors cause run-time errors. The effects of I/O errors are determined by the presence of *iocheck*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."
iocheck	An integer variable or integer array element that becomes defined as zero if no error is encountered, or as a positive integer if an error is encountered. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

■ Remarks

After writing the end-of-file record, the **ENDFILE** statement positions the file after the end-of-file record. Further sequential data transfer is prohibited unless you execute either a **BACKSPACE** or **REWIND** statement.

If a direct-access file is connected to *unitspec*, **ENDFILE** erases all records written beyond the new end-of-file record.

If a parameter of the **ENDFILE** statement is an expression that calls a function, that function must not cause an I/O statement or the **EOF** intrinsic function to be executed, because the results are unpredictable.

■ Example

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

5.3.19 The ENDIF Statement

■ Action

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

■ Syntax

ENDIF

■ Remarks

There must be a matching **ENDIF** statement for every block **IF** statement in a program unit. See Section 5.3.30, "The IF THEN ELSE Statement," for discussion and examples of block **IF** statements.

■ Example

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

5.3.20 The ENTRY Statement

■ Action

Specifies an entry point for a subroutine or external function

■ Syntax

ENTRY *ename*[[*[attrs]*]][[([[*formal*[[*[attrs]*]][[,*formal*[[*[attrs]*]]]...])]]

Parameter	Description
ename	The mandatory name of the entry point. If *ename* is an entry point for an external function, the type of *ename* must do one of the following: • Be declared in the external function's type-declaration section • Be typed by an **IMPLICIT** statement • Use default rules in determining type
attrs	An optional list of attributes separated by commas. The *attrs* describe *ename*. The following attributes can be used with *ename*: ALIAS, C, PASCAL, VARYING.
formal	An optional variable name, array name, or formal procedure name. If the **ENTRY** statement is in a subroutine, *formal* can be an asterisk.
attrs	A list of attributes separated by commas. The *attrs* describe *formal*. The following attributes can be used with *formal*: FAR, HUGE, NEAR, REFERENCE, VALUE.

■ Remarks

To begin executing a subroutine or function at the first executable statement after the **ENTRY** statement, do the following:

Type of Call	Form of Call
A subroutine	**CALL** *ename*[([*actual*[,*actual*]...])]
A function	*ename* ([*actual*[,*actual*]...])

Note that parentheses are required when calling a function, even if there are no arguments.

You can use *ename* to reference a subprogram in any program unit except the program unit that contains the **ENTRY** statement defining that name. That is, recursive references are not allowed.

There is no defined limit on the number of **ENTRY** statements you can use in a subprogram.

The following restrictions apply to use of the **ENTRY** statement:

- Within a subprogram, *ename* cannot be the same name used as a *formal* argument in a **FUNCTION**, **SUBROUTINE**, **ENTRY** or **EXTERNAL** statement.
- In a function subprogram, *ename* cannot appear in any statement other than a type statement until after *ename* has been defined in an **ENTRY** statement.
- If one *ename* in a function subprogram is of character type, all the *ename*s in that subprogram must be of character type, and all the *ename*s must be the same length.
- The argument *formal* cannot appear in an executable statement that occurs before the **ENTRY** statement in which *formal* appears, unless *formal* also appears in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement that precedes the executable statement.
- An **ENTRY** statement cannot be used between a block **IF** statement and the corresponding **ENDIF** statement, or between a **DO** statement and the terminal statement of its **DO** loop.

■ Example

```
C This program fragment writes a message
C indicating whether num is positive or negative
      IF (num.GE.0)CALL positive
      ELSE CALL negative
      .
      .
      .
      END
      SUBROUTINE sign
      ENTRY positive
      WRITE(*,*)'It''s positive.'
      RETURN
      ENTRY negative
      WRITE(*,*)'It''s negative.'
      RETURN
```

5.3.21 The EQUIVALENCE Statement

■ Action

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

■ Syntax

EQUIVALENCE (*nlist*) ‖,(*nlist*)‖...

Parameter	Description
nlist	A list of at least two variables, arrays, or array elements, separated by commas. An *nlist* may not include argument names. Subscripts must be integer constants and must be within the bounds of the array they index. No automatic type conversion occurs between the elements in *nlist*.

■ Remarks

An **EQUIVALENCE** statement specifies that the elements in *nlist* must have the same first memory location. Variables are said to be "associated" if they refer to the same actual memory location. Thus, an **EQUIVALENCE** statement associates the elements in *nlist*. If there is an array name in *nlist*, it refers to the first element of the array.

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:

```
¦01¦02¦03¦04¦05¦06¦07¦

¦-----A-----¦
      ¦-----B-----¦
¦--C(1)--¦--C(2)--¦
```

The following rules restrict how you may associate elements:

- A variable cannot be forced 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 CAUSES AN ERROR
      REAL R,S(10)
      EQUIVALENCE (R,S(1)),(R,S(2))
```

- Consecutive array elements must be stored in sequential order. The following, for example, is not permitted:

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

- Character and noncharacter elements cannot be associated when the $STRICT metacommand is in effect ($NOTSTRICT is the default).

- Character and noncharacter entities cannot be associated so that the noncharacter entities start on an odd byte boundary.

  Except in a common block, the compiler aligns noncharacter entities on word boundaries. The following, for example, causes 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, you must make sure that noncharacter elements are aligned at words. An error message is issued for any noncharacter elements that are not word aligned.

- An item that appears in *nlist* cannot be initialized in a type statement. The following example causes an error:

```
INTEGER i/1/
EQUIVALENCE (i,j)
```

- An **EQUIVALENCE** statement cannot cause sharing of memory between two different common blocks.

- An **EQUIVALENCE** statement can extend a common block by adding memory elements following the common block, as long as the **EQUIVALENCE** statement does not make a named common block's length different from the length of the same named common block in other program units.

- An **EQUIVALENCE** statement cannot extend a common block by adding memory elements preceding the common block, as in the following example:

```
C THIS CAUSES AN ERROR
      COMMON /ABCDE/ R(10)
      REAL S(10)
      EQUIVALENCE (R(1),S(10))
```

Unless the SSTRICT metacommand is specified, only the first subscript is required in EQUIVALENCE statements. This makes it easier to port FORTRAN 66 programs.

■ Example

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

5.3.22 The EXTERNAL Statement

■ Action

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

■ Syntax

EXTERNAL *name* [(*attrs*)][[,*name*[(*attrs*)]]...

Parameter	Description
name	The mandatory name of an external subroutine or function.
attrs	An optional list of attributes separated by commas. The *attrs* describe *name*. The following attributes can be used with *name*: ALIAS, C, FAR, NEAR, PASCAL, VARYING.

■ Remarks

Naming a subroutine or function in an **EXTERNAL** statement declares it as an external procedure. Statement-function names (of single-line functions) 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 FORTRAN, the **EXTERNAL** statement is used primarily to specify that a particular user-defined name is a subroutine or function to be used as a procedural parameter. The **EXTERNAL** statement may also indicate that a user-defined function is to replace an intrinsic function of the same name.

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

■ Examples

The following two program fragments contain examples of the **EXTERNAL** statement:

```
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)
```

5.3.23 The FORMAT Statement

■ Action

Directs the editing of data

■ Syntax

FORMAT (⟦*editlist*⟧)

Parameter	Description
editlist	A list of editing descriptions (for information on *editlist*, see Section 4.3.6).

■ Remarks

FORMAT statements must be labeled.

Invalid *editlist* strings generate warning messages.

Table 4.6 summarizes the nonrepeatable edit descriptors. Table 5.5 summarizes the repeatable edit descriptors. See Section 4.8, "Formatted I/O," for further information on edit descriptors and formatted I/O.

Table 5.5

Repeatable Edit Descriptors

Descriptor	Use
I*w*⟦**.***m*⟧	Integer editing
Z*w*	Hexadecimal editing
F*w***.***d*	Real editing
E*w***.***d*⟦**E***e*⟧	Real editing with exponent
G*w***.***d*⟦**E***e*⟧	Real editing for wide range of values
D*w***.***d*	Double-precision real editing
L*w*	Logical editing
A⟦*w*⟧	Character editing

5.3.24 The FUNCTION Statement (External)

■ Action

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

■ Syntax

[[*type*]] **FUNCTION** *fname* [[*fattrs*]] ([[*formal* [[*attrs*]]][[,*formal*[[*attrs*]]]]...)

Parameter	Description
type	Defines the type of the function. The *type* parameter must be one of the following: **CHARACTER** **CHARACTER*n** **COMPLEX** COMPLEX*8 COMPLEX*16 **DOUBLE PRECISION** **INTEGER** INTEGER*1 INTEGER*2 INTEGER*4 INTEGER[C] **LOGICAL** LOGICAL*1 LOGICAL*2 LOGICAL*4 **REAL** REAL*4 REAL*8 If *type* is omitted, the function's type is determined by any **IMPLICIT** or type statements that would determine the type of an ordinary variable. If *type* is specified, then the function name cannot appear in any type statements.

fname — The name of the function. The name *fname* cannot appear in **COMMON**, **DATA**, **EQUIVALENCE**, or **INTRINSIC** statements.

fattrs — A list of attributes, separated by commas. The *fattrs* describe *fname*. The following attributes can be used with *fname*: ALIAS, C, FAR, NEAR, PASCAL, VARYING.

formal — One or more formal argument names. If more than one is specified, they must be separated by commas.

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

attrs — A list of attributes separated by commas. The *attrs* describe *formal*. The following attributes can be used with *formal*: FAR, HUGE, NEAR, REFERENCE, VALUE.

■ Remarks

The function name is global, but it is also local to the function it names. The function name *fname* must appear as a variable in the program unit that defines the function. Every execution of that function must assign a value to *fname*. The final value of *fname*, 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 *fname* can be referenced in an expression, like any other variable.

In addition to returning the value of the function, an external function can return values by assignment to any of its formal arguments that were passed by reference.

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. Recursive calls are not detected by the compiler, even if they are direct.

There must be the same number of actual arguments in the **FUNCTION** statement as there are formal arguments in the corresponding function reference, unless the **C** and **VARYING** attributes have been used to declare the function.

■ Example

```
C EXAMPLE OF A FUNCTION REFERENCE
C GETNO IS A FUNCTION THAT READS A
C NUMBER FROM A FILE OPEN ON UNIT I
      I=2
   10 IF (GETNO(I) .EQ. 0.0) GOTO 10
      STOP
      END
C
      FUNCTION GETNO(NOUNIT)
      READ(NOUNIT, '(F10.5)') R
      GETNO = R
      RETURN
      END
```

5.3.25 The GOTO Statement (Assigned GOTO)

■ Action

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

■ Syntax

GOTO *variable* [[,] (*labels*)]

Parameter	Description
variable	An integer-variable name. When the assigned **GOTO** statement is executed, *variable* must have been assigned the label of an executable statement in the same program unit as the assigned **GOTO** statement.
labels	List of one or more statement labels of executable statements in the same program unit as the assigned **GOTO** statement. If more than one label is specified, the labels are separated by commas. The same *label* may appear repeatedly in the list of labels.

■ Remarks

If you specify the **$DEBUG** metacommand, a run-time error is generated if the label assigned to *name* is not one of the labels specified in the *label* list.

Jumping into a **DO**, **IF**, **ELSEIF**, or **ELSE** block from outside the block is not normally 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.

■ Examples

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

The following example uses an assigned **GOTO** statement to check the value of `clearance`:

```
$DEBUG

      integer   view, clearance
      .
      .
C These statements assign an appropriate label to view.
      IF (clearance .EQ. 1) THEN
            ASSIGN 200 TO view
         ELSEIF (clearance .EQ. 2) THEN
            ASSIGN 400 TO view
         ELSE
            ASSIGN 100 TO view
      ENDIF
      .
      .
C Show user appropriate view of data depending upon
C security clearance.
      GOTO view (100,200,400)

C If, in the above GOTO, view had not been assigned one
C of the valid labels of 100, 200, or 400, a run-time
C error would have been generated.

  100 CONTINUE
      .
      .
  200 CONTINUE
      .
      .
  400 CONTINUE
      .
      .
      END
```

5.3.26 The GOTO Statement (Computed GOTO)

■ Action

Transfers control to the statement at the *i*th label in the list

■ Syntax

GOTO (*labels*) [[,]] *i*

Parameter	Description
labels	One or more mandatory statement labels of executable statements from the same program unit as the computed **GOTO** statement. If more than one label is specified, the labels are separated by commas. The same statement label may be repeated in the list of labels.
i	A mandatory integer expression. Control is transferred to the *i*th label in the list.

■ Remarks

If there are *n* labels in the list of labels and *i* is out of range (that is, $i > n$ or $i < 1$), the computed **GOTO** statement acts like a **CONTINUE** statement. Otherwise, the next statement executed is the one at the *i*th *label* in the list of labels.

Jumping into a **DO**, **IF**, **ELSEIF**, or **ELSE** block from outside the block is not [illegible] permitted. A special feature, extended-range DO loops, does permit jumping into a DO block. See Section 6.2.3, "The sDelveMetacommand," for information.

■ Example

```
C EXAMPLE OF COMPUTED GOTO
      I = 1
C The following statement transfers control to
C statement 10
      GOTO (10, 20) I
      .
      .
   10 CONTINUE
      .
      .
   20 CONTINUE
```

5.3.27 The GOTO Statement (Unconditional GOTO)

■ Action

Transfers control to the statement specified by *label*

■ Syntax

GOTO *label*

Parameter	Description
label	The statement label of an executable statement in the same program unit as the **GOTO** statement.

■ Remarks

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

■ Example

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

5.3.28 The IF Statement (Arithmetic IF)

■ Action

Transfers control to the statement specified by one of three labels, depending on the result of *expression*

■ Syntax

IF (*expression*) *label1*, *label2*, *label3*

Parameter	Description
expression	An integer expression or a single- or double-precision real expression.
label1, *label2*, *label3*	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.

■ Remarks

The arithmetic **IF** statement transfers control as indicated in the following list:

If:	Then:
The *expression* < 0	The next statement executed is the statement at *label1*.
The *expression* $= 0$	The next statement executed is the statement at *label2*.
The *expression* > 0	The next statement executed is the statement at *label3*.

Jumping into a **DO**, **IF**, **ELSEIF**, or **ELSE** block from outside the block is not normally permitted. A special feature, extended-range **DO** loops, does permit jumping into a **DO** block. See Section 6.2.3, "The SDO66 Metacommand," for information.

■ Example

```
C EXAMPLE OF ARITHMETIC IF
      DO 40 J=-1,1
      I=J
C The following statement transfers control to
C statement 10 the first time, to statement 20 the
C second time, and to statement 30 the third time.
      IF (I) 10, 20, 30
   10 CONTINUE
      .
      .
   20 CONTINUE
      .
      .
   30 CONTINUE
      .
      .
   40 CONTINUE
```

5.3.29 The IF Statement (Logical IF)

■ Action

Executes *statement*, if *expression* is true; continues as if a **CONTINUE** statement were encountered, if *expression* is false

■ Syntax

IF (*expression*) *statement*

Parameter	Description
expression	A logical expression.
statement	Any executable statement except a **DO**, **ELSEIF**, **ELSE**, **ENDIF**, **END**, block **IF**, or another logical **IF** statement. Note that the statement can be an arithmetic **IF**.

■ Examples

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

5.3.30 The IF THEN ELSE Statement (Block IF)

■ Action

Begins executing statements in the **IF** block, if *expression* is true; transfers control to the next **ELSE**, **ELSEIF**, or **ENDIF** statement at the same **IF** level, if *expression* is false

■ Syntax

IF (*expression*) **THEN**

Parameter	Description
expression	A logical expression

■ Remarks

The associated **IF** block consists of all the executable statements (possibly none) between the **IF** statement and the next **ELSEIF**, **ELSE**, or **ENDIF** statement at the same **IF** level as this block **IF** statement.

The block **IF** statement transfers control as indicated in the following list:

If:	Then:
The *expression* is 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.
The *expression* is true, and there is at least one executable statement in the **IF** block	The next statement executed is the first executable statement in the **IF** block.
The *expression* is false	The next statement executed is the next **ELSEIF**, **ELSE**, or **ENDIF** statement at the same **IF** level as the 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 the block **IF** statement.

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

IF levels are defined as *if* minus *endif*, where *if* 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, and *endif* is the number of **ENDIF** statements from the beginning of the program unit up to, but not including, that statement.

Item	Required IF-Level Value
Statement	Greater than or equal to 0 and less than approximately 50
Block **IF**, **ELSEIF**, **ELSE**, and **ENDIF**	Greater than 0 and less than approximately 50
END statement	0

■ Examples

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

      IF(I.LT.10)THEN
C THE NEXT TWO STATEMENTS ARE ONLY EXECUTED IF
C I IS .LT. 10
        J=I
        SLICE=TAN(ANGLE)
      ENDIF

C Block IF with ELSEIF statements:

      IF(J.GT.1000)THEN
C Statements here are executed only if J.GT.1000
      ELSEIF(J.GT.100)THEN
C Statements here are executed only if J.GT.100 and
C J.LE.1000
      ELSEIF(J.GT.10)THEN
C Statements here are executed only if J.GT.10 and
C J.LE.100
      ELSE
C Statements here are executed only if J.LE.10
      ENDIF

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

      IF(I.LT.100)THEN
C Statements here are executed only if I.LT.100
        IF(J.LT.10)THEN
C Statements here are executed only if I.LT.100 and
C J.LT.10
        ENDIF
C Statements here are executed only if I.LT.100
      ELSE
C Statements here are executed only if I.GE.100
        IF(J.LT.10)THEN
C Statements here are executed only if I.GE.100 and
C J.LT.10
        ENDIF
C Statements here are executed only if I.GE.100
      ENDIF
```

5.3.31 The IMPLICIT Statement

■ Action

Defines the default type for user-declared names

■ Syntax

IMPLICIT *type* (*letters*) [[*type* (*letters*)]]...

Parameter	Description
type	One of the following types: **CHARACTER** **CHARACTER*** *n* **COMPLEX** COMPLEX*8 COMPLEX*16 **DOUBLE PRECISION** **INTEGER** INTEGER*1 INTEGER*2 INTEGER*4 INTEGER[C] **LOGICAL** LOGICAL*1 LOGICAL*2 LOGICAL*4 **REAL** REAL*4 REAL*8
letters	A list of single letters and ranges of letters. If more than one letter or range is listed, they must be separated by commas. 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. Note that Microsoft FORTRAN allows the use of the dollar sign ($) as an alphabetic character that follows the letter Z.

■ Remarks

An **IMPLICIT** statement defines the type and size for all user-defined names that begin with any of the letters specified. 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 the type indicated by 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)
C The following statement overrides the implicit
C INTEGER type for the variable ANYNAME
      CHARACTER*20 ANYNAME
      AGE = 10
      NAME = 'PAUL'
```

5.3.32 The INQUIRE Statement

■ Action

Examines the properties of a unit or a named file

■ Syntax

INQUIRE ({[**UNIT=**]*unitspec* | **FILE=***file*}
[,**ACCESS=***access*]
[,**BINARY=***binary*]
[,**BLANK=***blank*]
[,**BLOCKSIZE=***blocksize*]
[,**DIRECT=***direct*]
[,**ERR=***errlabel*]
[,**EXIST=***exist*]
[,**FORM=***form*]
[,**FORMATTED=***formatted*]
[,**IOSTAT=***iocheck*]
[,**MODE=***mode*]
[,**NAME=***name*]
[,**NAMED=***named*]
[,**NEXTREC=***nextrec*]
[,**NUMBER=***num*]
[,**OPENED=***opened*]
[,**RECL=***recl*]
[,**SEQUENTIAL=***seq*]
[,**SHARE=***share*]
[,**UNFORMATTED=***unformatted*])

Except as specified for *unitspec* below, the parameters can appear in any order.

Parameter	Description
unitspec	If **UNIT=** is omitted, *unitspec* must be the first specifier. You can use either an integer or an asterisk (*) as *unitspec*. If **UNIT=*** is specified, you may not include the **NUMBER=** option, or a run-time error is generated.

See Section 4.3.2, "Units," for information on unit specifiers. Exactly one *unitspec* or *file* must be specified, but not both. If *unitspec* is specified, the inquire operation is called an "inquire-by-unit" operation.

file

A character expression that evaluates to the name of the file being inquired about. Exactly one *unitspec* or *file* must be specified, but not both. If *file* is specified, the inquire operation is called an "inquire-by-file" operation.

access

A character variable or character array element. Set to **'SEQUENTIAL'** if the unit or file specified by *unitspec* or *file* is connected for sequential access. Set to **'DIRECT'** if the unit or file specified by *unitspec* or *file* is connected for direct access. In an inquire-by-unit operation, if no file is connected to *unitspec*, *access* is undefined.

binary

A character variable or character array element. Set to **'YES'** if binary is among the set of allowable forms for the file specified by *file* or connected to *unitspec*. Set to **'NO'** or **'UNKNOWN'** otherwise.

blank

A character variable or character array element. Set to **'NULL'** if the **BN** edit descriptor is in effect; set to **'ZERO'** if **BZ** is in effect.

blocksize

An integer variable or integer array element. If the *unitspec* or *file* is connected, *blocksize* is set to the I/O buffer size being used by the *unitspec* or *file*. If *unitspec* or *file* is not connected, *blocksize* is undefined.

direct

A character variable or character array element. Set to **'YES'** if direct is among the set of allowable access modes for the file specified by *file* or connected to *unitspec*. Set to **'NO'** or **'UNKNOWN'** otherwise.

errlabel — The label of an executable statement in the same program unit as the current **INQUIRE** statement. If *errlabel* is specified, I/O errors transfer control to the statement at *errlabel*. If *errlabel* is omitted, I/O errors cause run-time errors. The effects of I/O errors are determined by the presence of *iocheck*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

exist — A logical variable or logical array element. Set to **.TRUE.** if the unit or file specified by *unitspec* or *file* exists; set to **.FALSE.** otherwise.

form — A character variable or character array element. Set to **'FORMATTED'** if the unit or file specified by *unitspec* or *file* is connected for formatted I/O; set to 'BINARY' for binary I/O; or set to **'UNFORMATTED'** for unformatted I/O.

formatted — A character variable or character array element. Set to **'YES'** if formatted is among the set of allowable forms for the file specified by *file* or connected to *unitspec*; set to **'NO'** or **'UNKNOWN'** otherwise.

iocheck — An integer variable or integer array element that becomes defined as zero if no error is encountered, or as a positive integer if an error is encountered. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

mode — A character variable or character array element. Set to the current mode status of the specified unit. The strings are the same as those specified in the **OPEN** statement: 'READ', 'WRITE', and 'READWRITE'. In an inquire-by-unit operation, if no file is connected to *unitspec*, *mode* is undefined.

name

A character variable or character array element.

In an inquire-by-unit operation, set to the name of the file connected to *unitspec*. If no file is connected to *unitspec*, or if the file connected to *unitspec* does not have a name, *name* is undefined. In an inquire-by-file operation, set to the name specified for *file*.

named

A logical variable or a logical array element. Set to **.FALSE.** if the file specified by *file* or attached to *unitspec* is not open or if it is a scratch file; set to **.TRUE.** otherwise.

nextrec

An integer variable or integer array element that is assigned the record number of the next record in a direct-access file. The first record in a direct-access file has record number 1.

num

An integer variable or integer array element.

In an inquire-by-file operation, set to the number of the unit connected to *file*. If no unit is connected to *file*, *num* is undefined. In an inquire-by-unit operation, set to the number specified by *unitspec*. If you specify **UNIT = ***, do not include the **NUMBER=** option, or a run-time error is generated.

opened

A logical variable or logical array element.

In an inquire-by-unit operation, set to **.TRUE.** if any file is currently connected to *unitspec*; set to **.FALSE.** otherwise. In an inquire-by-file operation, set to **.TRUE.** if *file* is currently connected to any unit; set to **.FALSE.** otherwise.

recl

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

seq
: A character variable or character array element. Set to **'YES'** if sequential is among the set of allowable access modes for the file specified by *file* or connected to *unitspec*; set to **'NO'** or **'UNKNOWN'** otherwise.

share
: A character variable or character array element. Set to the current share status of the file specified by *file* or connected to *unitspec*. The strings are the same as those specified in the **OPEN** statement: **'COMPAT'**, **'DENYRW'**, **'DENYWR'**, **'DENYRD'**, and **'DENYNONE'**. In an inquire-by-unit operation, if no file is connected to *unitspec*, *share* is undefined.

unformatted
: A character variable or character array element. Set to **'YES'** if unformatted is among the set of allowable forms for the file specified by *file* or connected to *unitspec*; set to **'NO'** or **'UNKNOWN'** otherwise.

■ Remarks

The **INQUIRE** statement returns the values of the various attributes with which a file was opened. Note that the **INQUIRE** statement cannot determine the properties of an unopened file, and it cannot distinguish between attributes that were specified by you and attributes that were set by default.

You can execute the **INQUIRE** statement at any time. The values it returns are those that are current at the time of the call.

If a parameter of the **INQUIRE** statement is an expression that calls a function, that function must not cause an I/O statement or the **EOF** intrinsic function to be executed because the results are unpredictable.

■ Example

```
C This program asks for the name of a data file. The INQUIRE
C statement is then used to determine whether or not the file
C exists. If it does not, the program asks for another file
C name.

      CHARACTER*12     fname
      LOGICAL          exists

C Get the name of a file from the user.
 100  WRITE(*,'(1X,A)') 'Enter the name of the data file:  '
      READ(*,'(A12)') fname

C Get EXIST information about the specified file.
      INQUIRE(FILE=fname,EXIST=exists)

C Check to see if the specified file exists.
      IF (.not. exists) THEN
         WRITE(*,'(2A/)') ' *** Cannot find file ', fname
         GOTO 100
      ENDIF
C     .
C     .
      END
```

5.3.33 The INTEGER Statement

■ Action

Specifies the type of user-defined names

■ Syntax

INTEGER[[*bytes*]] [[C]] *vname*[[*attrs*]][[*length*]][[(*dim*)]][[/*values*/]]
[[,*vname*[[*attrs*]][[*length*]][[(*dim*)]][[/*values*/]]...

The order of the *length* and *dim* parameters can be reversed.

Parameter	Description
bytes	Must be 1, 2, or 4. The *bytes* parameter specifies the length, in bytes, of the items specified by the **INTEGER** statement. This value can be overridden by the *length* parameter.
vname	The symbolic name of a constant, variable, array, external function, statement function, or intrinsic function; or, a function subprogram or an array declarator. The *vname* parameter cannot be the name of a subroutine or main program.
attrs	A list of attributes separated by commas. The *attrs* describe *vname*. These attributes can be used with *vname*: **ALIAS**, **C**, **EXTERN**, **FAR**, **HUGE**, **NEAR**, **PASCAL**, **REFERENCE**, **VALUE**.
length	Must be 1, 2 or 4. The *length* parameter assigns the specified *length* to *vname*. If *length* is specified, it overrides the length attribute specified by *bytes*.
dim	A dimension declarator. You can only specify *dim* if *vname* is an array. If *dim* is specified, the **INTEGER** statement declares the array *vname*.

values A list of constants and repeated constants, separated by commas. A repeated constant is written in the form *n***constant*, where *n* is a positive-nonzero-integer constant, and is equivalent to *constant* repeated *n* times. The */values/* option, if specified, initializes *vname*. The following statement, for example, declares that num is of type **INTEGER**, and sets num equal to 10:

```
INTEGER num /10/
```

■ Remarks

An **INTEGER** statement confirms or overrides the implicit type of *vname*. The name *vname* is defined for the entire program unit, and cannot be defined by any other type statement in that program unit.

INTEGER statements must precede all executable statements.

■ Examples

```
C EXAMPLES OF INTEGER STATEMENTS
      INTEGER COUNT, MATRIX(4,4), SUM
      INTEGER*2 Q, M12*4, IVEC(10)*4, Z*4(10)
```

5.3.34 The INTERFACE Statement

■ Action

Declares a subprogram, its attributes, and formal argument types

■ Syntax

INTERFACE TO [*functionstatement* | *subroutinestatement*]

Parameter	Description
functionstatement	A function statement
subroutinestatement	A subroutine statement

■ Remarks

An interface defines the subroutine or function specified after the words **INTERFACE TO** in the interface.

An interface consists of an **INTERFACE** statement followed by the declaration statements of a subprogram. The only statements that can appear in an interface are **INTERFACE**, **EXTERNAL**, **INTRINSIC**, **DIMENSION**, **END**, and type statements.

The **INTERFACE** statement can be used to make sure that calls to a subprogram have appropriate arguments. If the subprogram and interface are in the same source file, the compiler ensures that the names, types, and number of arguments in the subprogram are consistent with those in the interface.

If you plan to compile parts of your program separately, you can include the interface in every source file that uses or defines the subprogram. The interface should occur before any references to the subprogram are made. If you put the text of the interface in a separate file and use the **$INCLUDE** metacommand (see Section 6.2.6, "The $INCLUDE Metacommand") in every file using the function or subroutine, then the same definition is used everywhere.

The compiler ensures that the arguments in calls to the subprogram are compatible with those defined by the interface. When the interface refers to a subprogram in the same source file, the compiler ensures that the names, types, and number of arguments are consistent.

Attributes used in an interface override the default definitions in the subprogram definition. However, if you use an attribute in the subprogram declaration or its arguments, the same attribute must appear in the **INTERFACE** statement.

■ Example

```
INTERFACE TO INTEGER FUNCTION f(I1,J1,K1)
INTEGER*2 I1
REAL J1
EXTERNAL K1
END
```

The above interface defines the following function:

```
INTEGER FUNCTION f(I,J,K)
INTEGER*2 I
REAL J
EXTERNAL K
.
.
.
END
```

5.3.35 The INTRINSIC Statement

■ Action

Declares that a name is an intrinsic function

■ Syntax

INTRINSIC *names*

Parameter	Description
names	One or more intrinsic-function names. If more than one name is specified, the names must be separated by commas.

■ Remarks

Each user-defined 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 Appendix B, "Intrinsic Functions."

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

■ Example

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

5.3.36 The LOCKING Statement

■ Action

Allows you to lock direct-access files and records to prevent access by other users in a network environment.

■ Syntax

LOCKING ([UNIT=]*unitspec*
[,ERR=*errlabel*]
[,IOSTAT=*iocheck*]
[,LOCKMODE=*lockmode*]
[,REC=*recc*]
[,RECORDS=*recnum*])

Except as specified for *unitspec* below, the parameters can appear in any order.

Parameter	Description
unitspec	An integer that is the number of the unit being locked. If **UNIT=** is omitted, *unitspec* must be the first parameter. The file attached to *unitspec* must have been opened for direct access (see Section 4.3.2, "Units," for more information on unit specifiers).
errlabel	The label of an executable statement in the same program unit as this statement. If *errlabel* is specified, I/O errors transfer control to the statement at *errlabel*. If *errlabel* is omitted, I/O errors cause run-time errors. The effects of I/O errors are determined by the presence of *iocheck*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."
iocheck	An integer variable or integer array element that becomes defined as zero if no error is encountered, or as a positive integer if an error is encountered. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

lockmode

A character expression with one of the following values:

Value	Description
'UNLCK'	Unlocks the specified region.
'LOCK'	Locks the specified region. Waits for any part of the region locked by a different process to become available.
'NBLCK'	Locks the specified region. If any of the records are already locked by a different process, an error is returned. This nonblocking lock is the default.
'RLCK'	Locks the specified region for write access only. This read lock is the same as **'LOCK'** except it locks for write access only.
'NBRLCK'	Locks the specified region for write access only. This nonblocking read lock is the same as **'NBLCK'** except it locks for write access only.

If you are using Microsoft FORTRAN with versions of MS-DOS® prior to 3.0, **'LOCK'**, **'RLCK'**, and **'NBRLCK'** are all identical to **'NBLCK'**.

rec

An integer expression that is the number of the first record in a group of records to be locked or unlocked. If *rec* is omitted, the next record (the one that would be read by a sequential read) is locked.

recnum

An integer expression that is the number of records to be locked. It defaults to 1.

■ Remarks

The **LOCKING** statement has no effect when used with versions of MS-DOS earlier than 3.0.

If a parameter of the **LOCKING** statement is an expression that calls a function, that function must not cause an I/O statement or the **EOF** intrinsic function to be executed, because the results are unpredictable.

5.3.37 The LOGICAL Statement

■ Action

Specifies the type of user-defined names

■ Syntax

LOGICAL [*bytes*] *vname*[*attrs*][* *length*][(*dim*)] [/ *values* /]
[,*vname*[*attrs*][* *length*][(*dim*)] [/ *values* /]]...

Parameter	Description
bytes	Must be 1, 2, or 4. The *bytes* parameter specifies the length, in bytes, of the items specified by the **LOGICAL** statement. This value can be overridden by the *length* parameter.
vname	The symbolic name of a constant, variable, array, external function, statement function, or intrinsic function; or, a function subprogram or an array declarator. The *vname* parameter cannot be the name of a subroutine or main program.
attrs	A list of attributes separated by commas. The *attrs* describe *vname*. These attributes can be used with *vname*: ALIAS, C, ENTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE.
length	Must be 1, 2, or 4. The *length* parameter assigns the *length* to *vname*. If *length* is specified, it overrides the length attribute specified by *bytes*. The default for *length* is the setting of the $STORAGE metacommand.
dim	A dimension declarator. If *dim* is specified, the **LOGICAL** statement declares *vname* as an array.

values A list of constants and repeated constants, separated by commas. A repeated constant is written in the form *n*constant*, where *n* is a positive-nonzero-integer constant, and is equivalent to *constant* repeated *n* times. The */values/* option, if specified, initializes *vname*. The following statement, for example, declares that `switch` is of type **LOGICAL**, and sets `switch` equal to `.TRUE.`:

```
LOGICAL switch /.TRUE./
```

■ Remarks

A **LOGICAL** statement confirms or overrides the implicit type of *vname*. The name *vname* is defined for the entire program unit, and it cannot be defined by any other type statement in that program unit.

LOGICAL statements must precede all executable statements.

■ Example

```
C EXAMPLE OF LOGICAL STATEMENTS
      LOGICAL SWITCH
```

5.3.38 The OPEN Statement

■ Action

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

■ Syntax

OPEN ([**UNIT** =]*unitspec*
[,**ACCESS** = *access*]
[,**BLANK** = *blank*]
[,**BLOCKSIZE** = *blocksize*]
[,**ERR** = *errlabel*]
[,**FILE** = *file*]
[,**FORM** = *form*]
[,**IOSTAT** = *iocheck*]
[,**MODE** = *mode*]
[,**RECL** = *recl*]
[,**SHARE** = *share*]
[,**STATUS** = *status*])

If the optional string **UNIT** = is omitted, then the parameter *unitspec* must be the first parameter. The other parameters can appear in any order.

Parameter	Description
unitspec	An external unit specifier. (See Section 4.3.2, "Units," for information about unit specifiers.)
access	A character expression; evaluates to **'DIRECT'**, when trailing blanks are removed, or to **'SEQUENTIAL'** (the default). These character constants may contain trailing blanks.
blank	A character variable or character array element. When trailing blanks are removed, blanks must be equal to **'NULL'** or **'ZERO'**. Set to **'NULL'** (for use with the **BN** edit descriptor); blank numeric fields in the file are ignored by default. If set to **'ZERO'** (for use with the **BZ** edit descriptor), blank numeric fields are loaded as zeros.

blocksize

An integer expression specifying the internal buffer size for use in I/O on the unit specified by *unitspec*. See Section 4.3.9, "Input/Output Buffer Size," for more information about internal buffer sizes.

errlabel

The label of an executable statement in the same program unit as this statement. If *errlabel* is specified, I/O errors transfer control to the statement at *errlabel*. If *errlabel* is omitted, I/O errors cause run-time errors. The effects of I/O errors are determined by the presence of *iocheck*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

file

A character expression. If *file* 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 file name specified is blank (**FILE=' '**), the following steps are taken:

1. The program attempts to read the name of the file from the list of names supplied (if any) on the command line used to invoke the program. If you specify a null argument on the command line (**" "**), you will be prompted for the corresponding file name. Execution of successive **OPEN** statements reads successive command-line arguments.
2. If there are more such **OPEN** statements than command-line arguments, the program prompts you for file names. Prompts are written to the unit associated with *.

For example, assume that the following command is used to invoke the program **MYPROG**:

```
myprog one " " two
```

MYPROG contains four **OPEN** statements with blank file names, in the following order:

```
OPEN(2, FILE=' ')
OPEN(4, FILE=' ')
OPEN(5, FILE=' ')
OPEN(10, FILE=' ')
```

Unit 2 is associated with file one. Since a null argument was specified on the command line for the second file name, the **OPEN** statement for unit 4 produces the following message:

```
File name missing or blank -
Please enter name UNIT 4?
```

Unit 5 is associated with file two. Since no fourth file was specified on the command line, the **OPEN** statement for unit 10 produces the following message:

```
File name missing or blank -
Please enter name UNIT 10?
```

form

A character expression that evaluates to **'FORMATTED'**, **'UNFORMATTED'**, or 'BINARY', when trailing blanks are removed.

If access is sequential, the default for *form* is **'FORMATTED'**; if access is direct, the default is **'UNFORMATTED'**.

iocheck

An integer variable or integer array element that becomes defined as a negative integer if an end-of-file record is encountered, as zero if no error is encountered, or as a positive integer if an error is encountered. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

mode

A character expression that evaluates to 'READ' (the process can read the file), **'WRITE'** (the process can write to the file), or **'READWRITE'** (the process can read and write to the file) when trailing blanks are removed.

The *mode* specifies the type of access to the file that will be made by the original process (the process that initially opened the file).

If you open a file without specifying *mode*, the FORTRAN run-time system always attempts to open with a *mode* value of **'READWRITE'**. If the open operation fails, the run-time system tries to open the file again, first using **'READ'**, then using **'WRITE'**. Note that this is not the same as

specifying **MODE='READWRITE'**. If you specify **MODE='READWRITE'**, and the file cannot be opened with both read and write access, the attempt to open the file fails. The default behavior (first trying **'READWRITE'**, then **'READ'**, then **'WRITE'**) is more flexible. Note that the value of the **STATUS=** option does not affect *mode*. See Section 4.3.11 for more information on file sharing.

recl

An integer expression that specifies the length of each record in bytes. This argument is required for direct-access files and ignored for sequential files.

share

A character expression. The acceptable values of *share* (when trailing blanks are removed) are the following:

Value	Description
'COMPAT'	Compatibility mode (the default mode). While a file is open in compatibility mode, the original user (the process that opened the file) may open the file in compatibility mode any number of times. No other user may open the file. A file that is already open in a mode other than compatibility mode cannot be opened in compatibility mode.
'DENYRW'	Deny-read write mode. While a file is open in deny-read write mode, no other process may open the file.
'DENYWR'	Deny-write mode. While a file is open in deny-write mode, no process may open the file with write access.

'DENYRD'	Deny-read mode. While a file is open in deny-read mode, no process may open the file with read access.
'DENYNONE'	Deny-none mode. While a file is open in deny-none mode, any process may open the file in any mode (except compatibility mode).

See Section 4.3.11 for information on file sharing.

status

A character expression that evaluates to **'OLD'**, **'NEW'**, **'UNKNOWN'**, or **'SCRATCH'**, when trailing blanks are removed.

The following list describes the four values of *status*:

Value	Description
'OLD'	If you specify **'OLD'**, the file must already exist. If you open a file with *status* equal to **'OLD'**, and the file does not already exist, an error is generated. See Section 4.3.10, "Error and End-of-File Handling," for information on handling I/O errors. If you open a file with *status* equal to **'OLD'**, and the file does already exist, the file is opened. If you open an existing (**'OLD'**) sequential file and write to it without first moving to the end of the file, you overwrite the file.
'NEW'	If you specify **'NEW'**, the file must not already exist. If you open a file with *status* equal to **'NEW'**, and the file does not already exist, the file is created. If you open a file with *status* equal to **'NEW'**, and the file does

already exist, an error is generated. See Section 4.3.10, "Error and End-of-File Handling," for information on handling I/O errors.

If you open a file with the option **STATUS='NEW'**, and subsequently close the file with **STATUS='KEEP'**, or if the program terminates without doing a close operation on the new file, a permanent file is created.

'SCRATCH' If no file name is specified when you open a file, the value of *status* is **'SCRATCH'**. Scratch files are temporary files. They are deleted when they are explicitly closed, or when the program terminates. You can also explicitly open a named file with the option **STATUS='SCRATCH'**. It will be deleted at program termination.

'UNKNOWN' If you open a file with the option **STATUS='UNKNOWN'**, the compiler first attempts to open the file with *status* equal to **'OLD'**, then with *status* equal to **'NEW'**.

The default is **STATUS='UNKNOWN'**. Using the status **'UNKNOWN'** allows you to avoid the run-time errors associated with opening an existing file with **STATUS='NEW'** or opening a nonexistent file with **STATUS='OLD'**.

Note that the status values affect only disk files. When a device such as the keyboard or the printer is opened as a file, the value of the **STATUS=** option does not matter.

■ Remarks

Opening a file for unit * has no effect, because unit * is permanently connected to the keyboard and screen. You can, however, use the **OPEN** statement to connect the other preconnected units (0, 5, and 6) to any unit.

If you do not provide an **OPEN** statement for a file, and the first operation using that file is a **READ** or **WRITE** operation, the program attempts to open a file as if a blank name were specified, as described with the *file* parameter. See Sections 5.3.43, "The READ Statement," and 5.3.52, "The WRITE Statement," for more information.

If a parameter of the **OPEN** statement is an expression that calls a function, that function must not cause an I/O statement or the EOF intrinsic function to be executed, because the results are unpredictable.

For more information on choosing values of *share* and *mode* when sharing files, see Section 4.3.11, "File Sharing."

■ Examples

The following example opens a new file:

```
C EXAMPLE 1
C PROMPT USER FOR A FILE NAME.
      CHARACTER*64 FNAME
      WRITE(*,'(A)')' Output file name?'
C READ THE FILE NAME FROM THE KEYBOARD.
      READ(*,'(A)') FNAME
C OPEN THE FILE FOR FORMATTED SEQUENTIAL ACCESS AND
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')
```

The following example opens an existing file:

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

5.3.39 The PARAMETER Statement

■ Action

Gives a constant a symbolic name

■ Syntax

PARAMETER (*name*=*expression* [,*name*=*expression*]...)

Parameter	Description
name	A symbolic name. The *name* must be of the same type as the *expression*. If, for example, you want to use the name `initial` for a real number, you must first declare `initial` as a real number, with a type or **IMPLICIT** statement. The largest allowed character constant is 1999 bytes.
expression	An expression. The *expression* can only include symbolic names if the symbolic names are defined in a previous **PARAMETER** statement in the current program unit.

■ Remarks

A symbolic name cannot be used for a complex number, in format specifications, or as a repeat count for an edit descriptor.

■ Examples

```
C EXAMPLE 1
      PARAMETER (NBLOCKS = 10)

C EXAMPLE 2
      REAL MASS
      PARAMETER (MASS = 47.3, pi=3.14159)

C EXAMPLE 3
      IMPLICIT REAL (L-M)
      PARAMETER (Loads = 10.0, MASS = 32.2)

C EXAMPLE 4
      CHARACTER*(*) BIG1
      PARAMETER(BIG1='This constant is 35 characters long')
```

5.3.40 The PAUSE Statement

■ Action

Temporarily suspends program execution and allows you to execute operating system commands.

■ Syntax

PAUSE [[*prompt*]]

Parameter	Description
prompt	Either a character constant or a number from 0 to 99,999.

■ Remarks

The default for *prompt* is the following:

```
"Please enter a blank line to continue or a system command."
```

The **PAUSE** statement suspends execution of the program. If a *prompt* is given, the prompt or number is displayed on the screen when the **PAUSE** statement is reached. The following list indicates possible responses to the prompt, and their results:

If:	Then:
The user enters a command	The command is executed and control is returned to the program.
The user enters a blank line	Control is returned to the program.
The user enters the word **COMMAND** (uppercase or lowercase)	The user can carry out a sequence of commands. To return control to the program, enter **EXIT** (uppercase or lowercase).

■ Example

```
C EXAMPLE OF A PAUSE STATEMENT
      SUBROUTINE setdrive (drive)
        PAUSE 'Please select default drive.'
        END
```

5.3.41 The PRINT Statement

■ Action

Specifies output to the screen

■ Syntax

PRINT *formatspec* [[,*iolist*]]

Parameter	Description
formatspec	A format specifier. For information on format specifiers, see Section 4.8, "Formatted I/O."
iolist	An I/O list specifying the data to be transferred. For information on I/O lists, see Section 4.3.8.

■ Remarks

The **PRINT** statement writes data to unit *.

If a parameter of the **PRINT** statement is an expression that calls a function, that function must not cause an I/O statement or the EOF intrinsic function to be executed, because the results are unpredictable.

■ Example

```
C The following two statements are equivalent:
      PRINT '(A11)','Abbottsford'
      WRITE(*,'(A11)')'Abbottsford'
```

5.3.42 The PROGRAM Statement

■ Action

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

■ Syntax

PROGRAM *programname*

Parameter	Description
programname	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.

■ Remarks

The main program is always assigned the default name of _main, in addition to any name specified by the PROGRAM statement.

■ Example

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

5.3.43 The READ Statement

■ Action

Transfers data from the file associated with *unitspec* to the items in the *iolist*, unless the end-of-file is reached or an error occurs

■ Syntax

READ {*formatspec*, |
([**UNIT=**]*unitspec*
[,[**FMT=**]*formatspec*]
[,**END=***endlabel*]
[,**ERR=***errlabel*]
[,**IOSTAT=***iocheck*]
[,**REC=***rec*])}
iolist

If the optional strings **UNIT=** and **FMT=** are omitted, then the parameters *unitspec* and *formatspec* must be the first and second parameters, respectively. The other parameters can appear in any order.

Parameter	Description
formatspec	A format specifier. The *formatspec* argument is required for a formatted read operation, and must not appear for an unformatted read operation. If a **READ** statement omits the **UNIT=**, **END=**, **ERR=**, and **REC=** options, and specifies only a *formatspec* and *iolist*, that statement reads from the asterisk unit (the keyboard). See Section 4.3.7 for information on format specifiers.
unitspec	A unit specifier. For an internal **READ** statement, *unitspec* is a character substring, character variable, character array element, character array, or noncharacter array. For an external read operation, *unitspec* is an external unit.

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

```
OPEN (unitspec,FILE=' ',
+   STATUS='OLD',
+   ACCESS='SEQUENTIAL',
+   FORM=form)
```

For the implicit open operation, *form* is **'FORMATTED'** for a formatted read operation or **'UNFORMATTED'** for an unformatted read operation. See Section 4.3.2, "Units," for information on unit specifiers.

endlabel

The label of a statement in the same program unit as the **READ** statement.

If *endlabel* is omitted, reading past the end-of-file record results in a run-time error. If *endlabel* is present, encountering the end-of-file record transfers control to the executable statement at *endlabel*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

errlabel

The label of an executable statement in the same program unit as this statement.

If *errlabel* is specified, I/O errors transfer control to the statement at *errlabel*. If *errlabel* is omitted, I/O errors cause run-time errors. The effects of I/O errors are determined by the presence of *iocheck*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

iocheck

An integer variable or integer array element that becomes defined as zero if no error is encountered, or as a positive integer if an error is encountered. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

rec

A positive integer expression, called the record number. It is specified for direct-access files only; if *rec* is given for a sequential-access or an internal file, an error results.

	The file is positioned to record number *rec* before transfer of data begins (the first record in the file has a record number of 1). The default for *rec* is the current position in the file.
iolist	The entities into which values are transferred from the file. For information on I/O lists, see Section 4.3.8.

■ Remarks

If the file was opened with **MODE='READWRITE'** (the default), you can alternately read and write to the same file without reopening it each time. You will usually need to rewind or backspace in a sequential file after writing.

Reading at or beyond the end-of-file record may cause an error. You cannot read unwritten records from a direct file.

Note that the data read from a binary file must correspond in size to the data written to a binary file.

If a parameter of the **READ** statement is an expression that calls a function, that function must not cause an I/O statement or the **EOF** intrinsic function to be executed, because the results are unpredictable.

■ 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(2I5,/,(8I5))
```

5.3.44 The REAL Statement

■ Action

Specifies the type of user-defined names

■ Syntax

REAL[(*bytes*)] *vname*[(*attrs*)][*****(*length*)][(*dim*)][/*values*/]
[,*vname*[(*attrs*)][*****(*length*)][(*dim*)][/*values*/]]...

The order of the *length* and *dim* parameters can be reversed.

Parameter	Description
bytes	Must be 4 or 8. The *bytes* parameter specifies the length, in bytes, of the items specified by the **REAL** statement. This value can be overridden by the *length* parameter.
vname	The symbolic name of a constant, variable, array, external function, statement function, or intrinsic function; or, a function subprogram or an array declarator. The *vname* parameter cannot be the name of a subroutine or main program.
attrs	A list of attributes separated by commas. The *attrs* describe *vname*. These attributes can be used with *vname*: **ALIAS**, **C**, **EXTERN**, **FAR**, **HUGE**, **NEAR**, **PASCAL**, **REFERENCE**, **VALUE**.
length	Must be 4 or 8. The *length* parameter assigns the specified *length* to *vname*. If *length* is specified, it overrides the length attribute specified by *bytes*.
dim	A dimension declarator. You can only specify *dim* if *vname* is an array. If *dim* is specified, the **REAL** statement declares the array *vname*.

values A list of constants and repeated constants, separated by commas. A repeated constant is written in the form *n*constant*, where *n* is a positive-nonzero-integer constant, and is equivalent to the constant *constant* repeated *n* times. The */values/* option, if specified, initializes *vname*. The following statement, for example, declares that num is of type **REAL**, and sets num equal to 0.0:

```
REAL num /0.0/
```

■ Remarks

A **REAL** statement confirms or overrides the implicit type of *vname*. The name *vname* is defined for the entire program unit, and it cannot be defined by any other type statement in that program unit.

REAL statements must precede all executable statements.

■ Examples

```
C EXAMPLES OF REAL STATEMENTS
      REAL GOOF,IABS
      REAL*4 WX1, WX3*4, WX5, WX8*4
```

5.3.45 The RETURN Statement

■ Action

Returns control to the calling program unit

■ Syntax

RETURN [[*ordinal*]]

Parameter	Description
ordinal	Defines an ordinal position for an alternate-return label in the formal argument list for the subroutine. See Section 5.3.5, "The CALL Statement," for information on alternate returns.

■ 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 function return variable. Execution of an **END** statement in a function or subroutine is equivalent to execution of a **RETURN** statement.

If the actual arguments of the **CALL** statement contain alternate-return specifiers, the **RETURN** statement can return control to a specific statement.

A **RETURN** statement in the main program is treated as a **STOP** statement with no *message* parameter (see Section 5.3.49, "The STOP Statement," for more information).

■ Examples

```
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 by the following statement
      END
```

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

```
   01 CALL AC(I,*10,J,*20,*30)
      WRITE (*,*) 'normal return'
      GOTO 40
   10 WRITE (*,*) 'I=10'
      GOTO 40
   20 WRITE (*,*) 'I=20'
      GOTO 40
   30 WRITE (*,*) 'I=30'
   40 CONTINUE
      .
      .
      .
      SUBROUTINE AC (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 1 specifies a return to the list's first alternate-return label, which is a symbol for the actual argument *10 in the CALL statement. RETURN 2 specifies a return to the second alternate-return label, and so on.

5.3.46 The REWIND Statement

■ Action

Repositions a file to its first record

■ Syntax

REWIND {*unitspec* |
(‖**UNIT** = ‖*unitspec*
‖,**ERR** = *errlabel*‖
‖,**IOSTAT** = *iocheck*‖)}

Parameter	Description
unitspec	An external unit specifier (see Section 4.3.2, "Units," for information about unit specifiers). If *unitspec* has not been opened, the **REWIND** statement has no effect.
errlabel	The label of an executable statement in the same program unit as the current statement. If *errlabel* is specified, I/O errors transfer control to the statement at *errlabel*. If *errlabel* is omitted, I/O errors cause run-time errors. The effects of I/O errors are determined by the presence of *iocheck*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."
iocheck	An integer variable or integer array element that becomes defined as zero if no I/O error is encountered, or as a positive integer if an error is encountered. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

■ Remarks

If a parameter of the **REWIND** statement is an expression that calls a function, that function must not cause an I/O statement or the **EOF** intrinsic function to be executed, because the results are unpredictable.

■ Example

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

5.3.47 The SAVE Statement

■ Action

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

■ Syntax

SAVE [*names*]

Parameter	Description
names	One or more names of common blocks (enclosed in slashes), variables, or arrays. If more than one name is specified, they must be separated by commas. After being saved, the named variables and all variables in the named common block have defined values if the current procedure is subsequently reentered. The default for *names* is the names of all allowable items in the current program unit.

■ Remarks

The **SAVE** statement does not allow the following:

- Redundant appearances of an item
- Formal argument names
- Procedure names
- Names of entities in a common block

A common block that is saved in one subprogram of an executable program must be saved in every subprogram that contains that common block in the executable program.

Note

Microsoft FORTRAN saves all variables by default. The **SAVE** statement is provided so that you can make your code portable.

■ Example

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

5.3.48 The Statement-Function Statement

■ Action

Defines a function in one statement

■ Syntax

fname ([[*formals*]]) = *expression*

Parameter	Description
fname	The name of the statement function. The name *fname* is local to the enclosing program unit and 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. If *fname* is used as a formal argument to another statement function, *fname* must have the same data type every time it is used.
formals	A list of formal-argument names. If there is more than one name, the names must be separated by commas. The list of formal-argument names defines 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 reused as other user-defined names in the rest of the program unit enclosing the statement-function definition. If *formal* is the same as another local name, then a reference to *formal* within the statement function always refers to the formal argument, never to the other local name. That local variable must be the same type as *formal*.

expression — Any expression.

Within the *expression*, 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.

■ Remarks

The statement-function statement defines the meaning of the statement function *fname*. The statement function *fname* can then be executed by a function reference in an expression.

The type of *expression* and the type of *fname* must be compatible in the same way that *expression* and *variable* must be compatible in assignment statements. For an explanation of assignment compatibility, see Section 2.7.1.2, "Type Conversion of Arithmetic Operands." The *expression* is converted to the same data type as *fname*. The actual arguments to the statement function are converted to the same type as the formal arguments.

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).

■ 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
```

5.3.49 The STOP Statement

■ Action

Terminates the program

■ Syntax

STOP [[*message*]]

Parameter	Description
message	Either a character constant or an integer from 0 to 99,999.

■ Remarks

If no *message* is specified, the following message is displayed on the screen when the program terminates:

```
STOP - Program terminated.
```

If a character constant is specified for *message*, that character constant is displayed on the screen when the program terminates. If a number is specified for *message*, then:

1. The words `Return code`, followed by the number, are printed on the screen when the program terminates.

   For example, see the following statement:

   ```
   STOP 0400
   ```

   The statement above produces the following output:

   ```
   Return code 0400
   ```

2. The program returns the least-significant byte of that integer value (a value from 0 to 255) to the operating system.

If a number is not specified for *message*, the program returns 0 to the operating system.

■ Example

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

5.3.50 The SUBROUTINE Statement

■ Action

Identifies a program unit as a subroutine, gives it a name, and identifies the formal arguments to that subroutine

■ Syntax

SUBROUTINE *subr* [[*satts*]] [[([[*formal* [[*fatts*]]] [[,*formal* [[*fatts*]]]]...])]]

Parameter	Description
subr	The user-defined, global, external name of the subroutine.
satts	A list of attributes, separated by commas. The *satts* describe *subr*. The following attributes can be used with *subr*: ALIAS, C, FAR, NEAR, PASCAL, VARYING
formal	The user-defined name of a formal argument. The formal argument may include the alternate return label (*). For an explanation of alternate return specifiers, see Section 5.3.5, "The CALL Statement."
fatts	A list of attributes separated by commas. The *fatts* describe *formal*. The following attributes can be used with *formal*: FAR, HUGE, NEAR, REFERENCE, VALUE

■ Remarks

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

In a **CALL** statement, actual arguments that reference a subroutine must agree with corresponding formal arguments in the **SUBROUTINE** statement, in order, in number except when the C and VARYING attributes are specified, and in type or kind.

The compiler will check for correspondence if the formal arguments are known. See Section 5.3.5, "The CALL Statement," for information on checking arguments for correspondence.

■ Example

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

5.3.51 The Type Statements

See individual listings in this chapter for the **CHARACTER**, **COMPLEX**, **DOUBLE PRECISION**, **INTEGER**, **LOGICAL**, and **REAL** statements.

5.3.52 The WRITE Statement

■ Action

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

■ Syntax

WRITE (⟦**UNIT=**⟧*unitspec*
⟦,⟦**FMT=**⟧*formatspec*⟧⟧
⟦,**ERR=***errlabel*⟧⟧
⟦,**IOSTAT=***iocheck*⟧⟧
⟦,**REC=***rec*⟧⟧)
iolist

The parameters may appear in any order, except as described for *unitspec* and *formatspec* below.

Parameter	Description

unitspec	A unit specifier.
	
	If the optional string **UNIT=** is omitted, *unitspec* must be the first parameter. See Section 4.3.2, "Units," for information about unit specifiers.
	
	For an internal write operation, *unitspec* must be a character substring, character variable, character array element, character array, or noncharacter array. For an external write operation, *unitspec* must be an external unit.
	
	If the file associated with *unitspec* has not been opened by an **OPEN** statement, an implicit open operation, equivalent to the following statement, is performed:
	
	`OPEN (unitspec, FILE=' ',`
	`+ STATUS='UNKNOWN',`
	`+ ACCESS='SEQUENTIAL',`
	`+ FORM=form)`

Here, *form* is 'FORMATTED' for a formatted WRITE statement and 'UNFORMATTED' for an unformatted WRITE statement.

formatspec

A format specifier.

If the optional string **FMT=** is omitted, then *formatspec* must be the second parameter. This argument must not appear for an unformatted write operation; it is required for a formatted write operation. See Section 4.3.7 for information on format specifiers.

errlabel

The label of an executable statement in the same program unit as the current statement.

If *errlabel* is specified, I/O errors transfer control to the statement at *errlabel*. If *errlabel* is omitted, I/O errors cause run-time errors. The effects of I/O errors are determined by the presence of *iocheck*. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

iocheck

An integer variable or integer array element that becomes defined as zero if no error is encountered, or as a positive integer if an error is encountered. For more information on error handling, see Section 4.3.10, "Error and End-of-File Handling."

rec

A positive integer expression, called a record number, specified for direct-access and binary files only (otherwise, an error results).

The argument *rec* specifies the number of the record to be written. The first record in the file is record number 1. The default for *rec* is the current position in the file.

iolist

A list of entities whose values are transferred by the **WRITE** statement.

For information on I/O lists, see Section 4.3.8.

■ Remarks

If the file was opened with **MODE='READWRITE'** (the default), you can alternately read and write to the same file without reopening it each time. You will usually need to rewind or backspace in a sequential file after writing.

Reading at or beyond the end-of-file record may cause an error. You cannot read unwritten records from a direct file.

If you write to a sequential file, you delete any records that existed beyond the newly written record. Note that for sequential files, you are always effectively at the end of the file following a write operation, and you must backspace or rewind before the next read operation. Also, of course, the operating system can deny access to a file.

If a parameter of the **WRITE** statement is an expression that calls a function, that function must not cause an I/O statement or the **EOF** intrinsic function to be executed, because the results are unpredictable.

■ Example

```
C Generate a table of square and cube roots
C of the whole numbers from 1-100
      WRITE(*,10)(I,SQRT(FLOAT(I)),FLOAT(I)**(1./3.),I=1,100)
   10 FORMAT(I5,F8.4,F8.5)
      END
```

Chapter 6
Metacommands

6.1 Introduction

The first part of this chapter lists the metacommands available in Microsoft FORTRAN. The second part of the chapter contains a directory of FORTRAN metacommands, listed alphabetically.

Metacommands give the Microsoft FORTRAN Compiler information on how to process source code. The *Microsoft FORTRAN Compiler User's Guide* contains additional information on using metacommands. Note that command-line options, as described in the *Microsoft FORTRAN Compiler User's Guide*, can be used for the same purposes as many of the metacommands.

Table 6.1 summarizes the Microsoft FORTRAN metacommands. Note that many of the metacommands can be turned on or off by specifying the metacommand or its opposite, such as **$DECLARE** or **$NODECLARE**.

Table 6.1
Metacommands

Metacommand	Instructions to Compiler	Default
$DEBUG[**:***string*]	Turns on run-time checking for integer arithmetic operations, assigned **GOTO** values, subscript bounds, and substrings. **$NODEBUG** turns off checking. **$DEBUG** does not trigger or suppress floating-point exceptions. **$DEBUG** can also be used for conditional compilation.	**$NODEBUG**
$DECLARE	Generates warning messages for undeclared variables. **$NODECLARE** turns off these messages.	**$NODECLARE**
$DO66	Uses FORTRAN 66 semantics for **DO** statements.	**$DO66** not set
$FLOATCALLS	Generates calls to subroutines in the emulator library. **$NOFLOATCALLS** causes the compiler to generate in-line interrupt instructions.	**$NOFLOATCALLS**
$FREEFORM	Uses free-form format for source code. **$NOFREEFORM** uses fixed format.	**$NOFREEFORM**
$INCLUDE:'*filename*'	Proceeds as if *filename* were inserted at this point in the current source file.	None
$LARGE[**:***name* [**,***name*]]	Addresses the named array outside of the **DGROUP** segment. **$NOTLARGE** disables **$LARGE** for the named array. If *name* is omitted, these metacommands affect all arrays.	None
$LINESIZE:*n*	Makes subsequent pages of listing *n* columns wide. Minimum *n* equals 40; maximum *n* equals 132.	**$LINESIZE:80**

Table 6.1 *(continued)*

Metacommand	Instructions to Compiler	Default
$LIST	Begins generation of listing information that is sent to the listing file. **$NOLIST** suppresses generation of listing information.	**$LIST**
$MESSAGE:*string*	Sends a character string to the standard output device.	None
$PAGE	Starts new page of listing.	None
$PAGESIZE:*n*	Makes subsequent pages of listing *n* lines long. Minimum *n* equals 15.	**$PAGESIZE:63**
$STORAGE:*n*	Allocates *n* bytes of memory (2 or 4) to all **LOGICAL** or **INTEGER** variables.	**$STORAGE:4**
$STRICT	Disables Microsoft FORTRAN features not in 1977 full-language standard. **$NOTSTRICT** enables them.	**$NOTSTRICT**
$SUBTITLE:*subtitle*	Uses subtitle *subtitle* for subsequent pages of listing.	**$SUBTITLE:"C**
$TITLE:*title*	Uses title *title* for subsequent pages of listing.	**$TITLE:"C**
$TRUNCATE	Truncates variables to six characters. **$NOTRUNCATE** turns off truncation.	**$TRUNCATE**

Any line with a **$** character in column 1 is interpreted as a metacommand. A metacommand and its arguments (if any) must fit on a single source line; continuation lines are not permitted.

6.2 Metacommand Directory

The remainder of this chapter is an alphabetical directory of available Microsoft FORTRAN metacommands. Each metacommand is described using the following format:

Heading	Information
■ Action	Summary of what the metacommand does.
■ Syntax	Correct syntax for the metacommand, and description of the metacommand's parameters.
■ Remarks	Use of the metacommand.
■ Example	Sample programs or program segments that illustrate the use of the metacommand. This section does not appear with every reference entry.

6.2.1 The $DEBUG and $NODEBUG Metacommands

■ Action

$DEBUG directs the compiler to perform additional testing and expanded error handling. **$DEBUG** can also be used for conditional compilation. **$NODEBUG** suppresses the additional testing and expanded error handling.

■ Syntax

$[NO]DEBUG[*:string*]

■ Remarks

The default is **$NODEBUG**.

These metacommands can appear anywhere in a program.

When **$DEBUG** is set, the compiler does the following:

- Tests integer arithmetic for overflow.
- Tests assigned **GOTO** values against the optional label list in an assigned **GOTO** statement.
- Provides the run-time error-handling system with file names and line numbers. If any run-time errors occur, the file name and line number are displayed on the console.
- Checks range of substrings.
- Checks assignment range. This catches errors when larger integer variables are assigned to smaller integer variables, such as assigning an **INTEGER*4** variable to an **INTEGER*2** variable. If **$DEBUG** is not set, the variable is truncated, no error is reported, and the program returns unpredictable results. In the case of real numbers, an error is always reported.

Note

$DEBUG does not trigger or suppress floating-point exception handling. Microsoft FORTRAN conforms to the proposed IEEE standard for exception handling for the five following conditions: invalid operation, division by zero, overflow, underflow, and precision. See the *Microsoft FORTRAN Compiler User's Guide* for information on exception handling on your system.

This metacommand should be placed into each source file to be compiled.

If the optional *string* is specified, the characters in *string* specify that lines that have those characters in column 1 are to be compiled into the program. Case is not significant. Note that the letter C always indicates a comment line; therefore, if *string* contains a C, the C is ignored. If more than one **$DEBUG:***string* metacommand is specified, each *string* overrides the previous *string*. Note that **$DEBUG** can be used for conditional compilation only if the **$FREEFORM** metacommand has not been specified. If the **$DEBUG:***string* metacommand is specified after **$FREEFORM**, a warning message is produced.

■ Example

```
C If the $FREEFORM metacommand has been specified, the
C next line produces an error message.
$DEBUG:'ABCD'
A         I=1
E         I=2
B         I=I+I
F         I=I*I
C This is always a comment. I equals 2,
C because only statements A and B are executed.
```

6.2.2 The $DECLARE and $NODECLARE Metacommands

■ Action

$DECLARE generates warnings for undeclared variables. **$NODECLARE** does not.

■ Syntax

$[NO]DECLARE

■ Remarks

The default is **$NODECLARE**. When **$DECLARE** is set, a warning message is generated at the first use of any variable that has not been declared in a specification statement.

■ Example

```
$DECLARE
C Since the variable z never appears in a type
C statement, its use in the statement labeled 100
C produces an error.
C
      REAL x,y
      y = 1.0
C The following statement produces an error:
100   x=y+z
      END
```

6.2.3 The $DO66 Metacommand

■ Action

$DO66 causes DO statements to conform to FORTRAN 66 semantics.

■ Syntax

$DO66

■ Remarks

You must obey the following rules when using **$DO66**:

- **$DO66** must precede the first declaration or executable statement of the source file in which it occurs.
- **$DO66** may only be preceded by a comment line or another metacommand.
- **$DO66** may only appear once in the source file.

When **$DO66** is on, the following FORTRAN 66 semantics are used:

- All **DO** statements are executed at least once.
- 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 include, logically, 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.

Note how this differs from the default (FORTRAN 77) semantics, which are as follows:

- **DO** statements can be executed zero times, if the value of the initial control variable exceeds that of the final control variable (or the corresponding condition for a **DO** statement with negative increment).

- 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 Metacommands

■ Action

$FLOATCALLS causes floating-point operations to be processed by calls to library subroutines. **$NOFLOATCALLS** causes floating-point operations to be processed by compiler-generated, in-line coprocessor instructions.

■ Syntax

$[NO]FLOATCALLS

■ Remarks

$NOFLOATCALLS is the default.

See the *Microsoft FORTRAN Compiler User's Guide* for a discussion of the advantages and disadvantages of each method.

■ Example

```
$FLOATCALLS
        REAL X, SINE

        WRITE (*,100)
 100    FORMAT(1X,'ENTER X: ')
        READ (*,'(F10.5)') X

        WRITE (*,200) X, SINE(X, .00001)
 200    FORMAT (1X, 'THE SINE OF ', F10.5, ' = ', F9.6)
        END

        REAL FUNCTION SINE(X, EPS)
C The function calculates the sine of X using a power
C series. Successive terms are calculated until less
C than eps.

C Library calls are generated instead of in-line
C instructions, allowing this routine to be run on
C machines with or without a numeric coprocessor.

            REAL X, Y, Z, NEXT, I, EPS

            Z = AMOD(X, 6.2831853)
            Y = Z
            I = 4.0
            NEXT = -Z*Z*Z / 6.0
 100        IF (ABS(NEXT) .GE. EPS) THEN
                Y = Y + NEXT
                NEXT = -NEXT*Z*Z / (I*(I+1.0))
                I = I + 2.0
                GOTO 100
            ENDIF
            SINE = Y
            RETURN

        END
```

6.2.5 The $FREEFORM and $NOFREEFORM Metacommands

■ Action

$NOFREEFORM specifies that a source file is in the standard FORTRAN format. **$FREEFORM** specifies that a source file is in the free-form format.

■ Syntax

$[NO]FREEFORM

■ Remarks

If this metacommand appears, it must precede any FORTRAN statements. The default, **$NOFREEFORM**, tells the compiler that your source code is in the standard FORTRAN format: labels are in columns 1–5, continuation characters are in column 6, statements are in columns 7–80, and characters beyond column 80 are ignored. The standard FORTRAN format is described in Section 3.2, "Lines." **$FREEFORM** tells the compiler that your source code is in the following format:

- A double quotation mark (") in column 1 indicates a comment line.
- Initial lines may start in any column.
- The first nonblank character of an initial line may be a digit: the first digit in a statement number. The statement number may be from one to five decimal digits; blanks and leading zeros are ignored. Blanks are not required to separate the statement number from the first character of the statement.
- If the last nonblank character of a line is a minus sign, it is discarded and the next line is taken to be a continuation line. The continuation line may start in any column.
- Alphabetic characters and asterisks are not allowed as comment markers in column 1.

■ Example

```
$FREEFORM

"The sine of the entered number X is calculated using a
"power series. Successive terms are calculated until
"one is less than EPSILON.

REAL X, EPSILON, Z, SINE, NEXT
EPSILON = 0.0001

  WRITE (*,1)
1 FORMAT (1X,'ENTER X: ')
  READ (*,'(F10.5)') X

  Z    = AMOD(X, 6.2831853)
  SINE = Z
  I    = 4.0
  NEXT = -Z*Z*Z / 6.0
2 IF (ABS(NEXT) .GE. EPSILON) THEN
     SINE = SINE + NEXT
     NEXT = -NEXT*Z*Z / (I*(I+1.0))
     I = I + 2.0
     GOTO 2
  ENDIF

  WRITE (*, 3) X, SINE
3 FORMAT (1X, 'THE SINE OF ', F10.5, -
          ' = ', F12.10)
END
```

6.2.6 The $INCLUDE Metacommand

■ Action

$INCLUDE directs the compiler to proceed as though the **$INCLUDE** metacommand were replaced by the file specified by the metacommand.

■ Syntax

$INCLUDE:'*filename*'

■ Remarks

The argument *filename* must be a valid file specification for your operating system.

At the end of the included file, the compiler resumes processing the original source file at the line following the **$INCLUDE** metacommand.

Included files can also contain **$INCLUDE** metacommands; this is called "nesting" included files. The compiler allows you to nest up to ten **$INCLUDE** metacommands; your operating system may impose further restrictions.

The **$INCLUDE** metacommand is particularly useful for guaranteeing that several modules use the same declaration for a common block, and for using the **INTERFACE** statement to ensure consistency in subroutine and function argument lists.

■ Example

This program implements a stack by declaring the common stack data in an include file. The contents of the file **STKVARS.FOR** (shown below the following program) are substituted into the source code for every **$INCLUDE** metacommand. This guarantees that all references to common storage for stack variables are consistent.

```
      integer i
      real    x
$INCLUDE:'stkvars.for'

C Read in five real numbers.
      DO 100 i = 1,5
         READ(*,'(f10.5)') x
         CALL push(x)
 100  CONTINUE

C     Write the numbers out in reverse order.
      WRITE(*,*) ' '
      DO 200 i = 1,5
         CALL pop(x)
         WRITE(*,*) x
200   CONTINUE
      END


      SUBROUTINE push(x)
C Subroutine pushes an element X onto the top of the
C stack.

         REAL x
$INCLUDE:'stkvars.for'

         top = top+1
         IF(top .GT. stacksize) STOP 'Stack overflow'
         stack(top) = x
         RETURN
      END

      SUBROUTINE pop(x)
C Subroutine pops an element from the top of the stack
C into X.

      REAL x
$INCLUDE:'stkvars.for'

         IF (top .LE. 0) STOP 'Stack underflow'
         x = stack(top)
         top = top-1
         RETURN
      END
```

The following is the file **STKVARS.FOR**:

```
C This file contains the declaration of the common
C block for a stack implementation. Because this file
C contains an assignment statement, it must be included
C only after all other specification statements in
C each program unit.

      REAL       stack(500)
      INTEGER    top, stacksize

      COMMON     /stackbl/ stack, top

      stacksize = 500
```

6.2.7 The $LARGE and $NOTLARGE Metacommands

Action

$LARGE specifies that an actual argument can span more than one segment (64K). **$NOTLARGE** specifies that an actual argument cannot span more than one segment (64K).

Syntax

$[NOT]LARGE[:*names*]

Parameter	Value
names	One or more names of array variables or formal array arguments. If more than one name is specified, they must be separated by commas. When *names* is specified in the **$LARGE** metacommand, it indicates that the array or formal array argument specified can span more than one segment (that is, it is addressed outside of **DGROUP**). When *names* is specified in the **$NOTLARGE** metacommand, it excludes the specified items from the effects of a **$LARGE** metacommand used without arguments.

Remarks

$NOTLARGE is the default.

If the optional *names* parameter is specified, the metacommand must occur in the declarative section of a subprogram.

If *names* is omitted, the metacommand affects all arrays in all subsequent subprograms in the source file until the **$NOTLARGE** metacommand is specified without any arguments. This form may occur anywhere except in the executable section of a subprogram.

Arrays with explicit dimensions indicating that they are bigger than 64K are automatically allocated to multiple segments outside of the default data segment. You do not need to specify **$LARGE** for these arrays.

Only one **$LARGE** or one **$NOTLARGE** metacommand without arguments can occur in a single program unit. The following code fragment, for example, is illegal:

```
C This is illegal:
$LARGE
      SUBROUTINE mysub
$NOTLARGE
      a=1.0
       .
       .
```

Note that use of the **$LARGE** metacommand on the entire program corresponds to the "huge" memory model. You can also use the **HUGE** attribute to specify that an actual argument can span more than one segment.

6.2.8 The $LINESIZE Metacommand

■ Action

$LINESIZE formats subsequent pages of the listing to a width of *n* columns.

■ Syntax

$LINESIZE:*n*

Parameter	Value
n	A positive integer between 80 and 132. The default for *n* is 80.

■ Example

```
$LINESIZE:100

C The compiler listing of this program will be one
C hundred columns wide.

      REAL     x
      x = 20
      WRITE (*,100) x, sqrt(x)
100   FORMAT(' The square root of ',f5.2,' is ',f7.4)
      END
```

6.2.9 The $LIST and $NOLIST Metacommands

- Action

$LIST sends subsequent listing information to the listing file specified when starting the compiler. **$NOLIST** directs that subsequent listing information be discarded, until there is a subsequent occurrence of the **$LIST** metacommand.

- Syntax

$NOLIST

- Remarks

The default is $LIST.

If no listing file is specified in response to the compiler prompt, the metacommand has no effect.

$LIST and $NOLIST can appear anywhere in a source file.

6.2.10 The $MESSAGE Metacommand

Action

$MESSAGE sends a character string to the standard output device during the first pass of the compiler.

Syntax

$MESSAGE:*string*

Parameter	Description
string	A character constant

Remarks

The specified string is sent to the standard output device during the first pass of the compiler.

Example

The following example writes `Compiling program` to the standard output device during execution of the first pass of the compiler:

```
$MESSAGE: 'Compiling program'
```

6.2.11 The $PAGE Metacommand

■ Action

$PAGE starts a new page in the source-listing file.

■ Syntax

$PAGE

■ Remarks

If the first character of a line of source text is the ASCII form-feed character (hexadecimal code 0C), it is the same as having a **$PAGE** metacommand before that line.

■ Example

```
C This is page one.
C The following metacommand will start a new page in
C the source listing file:
$PAGE
C This is page two.
C The following line starts with the ASCII form-feed
C character, so it will also start a new page in the
C source listing file:

C This is page 3.
      STOP ' '
      END
```

6.2.12 The $PAGESIZE Metacommand

- **Action**

$PAGESIZE formats subsequent pages of the source listing to a length of *n* lines.

- **Syntax**

$PAGESIZE:*n*

- **Remarks**

The argument *n* must be at least 15. The default page size is 63 lines.

6.2.13 The $STORAGE Metacommand

Action

$STORAGE allocates *n* bytes of memory for all variables declared in the source file as integer or logical variables.

Syntax

$STORAGE:*n*

Remarks

The argument *n* must be either 2 or 4. The default is 4.

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 Microsoft FORTRAN **$STORAGE** metacommand to a value of 2, programs will default to 32-bit arithmetic and may run more slowly than expected. Setting the **$STORAGE** metacommand to 2 allows programs to run faster and to be smaller.

The **$STORAGE** metacommand does not affect the allocation of memory for variables declared with an explicit length specification, such as **INTEGER*2** or **LOGICAL*4**.

If several files of a source program are compiled and linked together, be 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.

The default allocation for **INTEGER**, **LOGICAL**, and **REAL** variables is 4 bytes. This results in **INTEGER**, **LOGICAL**, and **REAL** variables being allocated the same amount of memory, as required by the FORTRAN-77 standard.

For information on how the **$STORAGE** metacommand affects arithmetic expressions, see Section 2.7.1.2, "Type Conversion of Arithmetic Operands." For information on how the **$STORAGE** metacommand affects the passing of integer arguments, see Section 3.6, "Arguments."

Example

```
$STORAGE:2

C Since b and c are declared as integers and storage is
C set to 2 bytes, they will only be two bytes long.
C a and d will be 4 bytes long.

      integer*4 a, d
      integer   b, c

      a = 65537
      b = 1024

C Since c is only 2 bytes, it will get the lower two
C bytes of a+b.
      c = a+b
      d = a+b

C The following statement produces:  1025        66561
      write(*,*) c, d

      end
```

6.2.14 The $STRICT and $NOTSTRICT Metacommands

■ Action

$STRICT disables the specific Microsoft FORTRAN features not found in the FORTRAN 77 full-language standard, and **$NOTSTRICT** enables these features.

■ Syntax

$[NOT]STRICT

■ Remarks

The default is **$NOTSTRICT**.

Note that all features in Microsoft FORTRAN that are not in the FORTRAN 77 full-language standard are printed in blue in this manual.

$STRICT and **$NOTSTRICT** can appear anywhere in a source file.

■ Example

```
$STRICT
C The following statement produces an error, because
C INTEGER*2 is not part of the FORTRAN 77 full-language
C standard:
      INTEGER*2        i
C The variable balance will be truncated to six
C characters:
      REAL             balance(500)
C The following statement produces an error, because
C the MODE= option is not part of the FORTRAN 77 full-
C language standard:
      OPEN(2, FILE='BALANCE.DAT',MODE='READ')

      DO 100 i = 1, 500
C The following statement produces an error, because
C the EOF intrinsic function is not part of the
C FORTRAN 77 full-language standard (EOF is treated
C as a real function):

         IF (EOF(2)) GOTO 200
         READ(2,'(F7.2)') balance(i)
 100  CONTINUE
 200  CONTINUE

C     .
C     .

      END
```

6.2.15 The $SUBTITLE Metacommand

■ Action

$SUBTITLE assigns the specified subtitle for subsequent pages of the source listing.

■ Syntax

$SUBTITLE:*subtitle*

Parameter	Description
subtitle	Any valid character constant

■ Remarks

If a program contains no **$SUBTITLE** metacommand, the subtitle is a null string. The value of the *subtitle* string is printed in the upper left corner of the source-listing file pages, below the title, if any. Note that, for a subtitle to appear on a specific page of the source-listing file, the **$SUBTITLE** metacommand must either be the first statement on that page, or must have appeared on a previous page.

If more than one **$SUBTITLE** metacommand is specified, earlier subtitles are overridden by later ones.

■ Example

The following program produces a listing in which each page is titled **GAUSS** (the name of the program). Each subprogram begins on a new page of the listing, and the name of the subprogram appears as a subtitle.

```
$TITLE:'GAUSS'

C Main program here...
          END

$SUBTITLE:'Row Division'
$PAGE
          SUBROUTINE divide(row, matrix, pivot)
C Subroutine body...
            RETURN
          END

$SUBTITLE:'Back Substitution'
$PAGE
          SUBROUTINE backsub(matrix)
C Subroutine body...
            RETURN
          END
```

6.2.16 The $TITLE Metacommand

■ Action

$TITLE assigns the specified title for subsequent pages of the source listing (until overridden by another **$TITLE** metacommand).

■ Syntax

$TITLE:*title*

Argument	Description
title	Any valid character constant

■ Remarks

If a program contains no **$TITLE** metacommand, the title is a null string.

■ Example

The following program produces a listing in which each page is titled **GAUSS** (the name of the program). Each subprogram begins on a new page of the listing, and the name of the subprogram appears as a subtitle.

```
$TITLE:'GAUSS'

C Main program here...
        END

$SUBTITLE:'Row Division'
$PAGE
        SUBROUTINE divide(row, matrix, pivot)
C Subroutine body...
           RETURN
        END

$SUBTITLE:'Back Substitution'
$PAGE
        SUBROUTINE backsub(matrix)
C Subroutine body...
           RETURN
        END
```

6.2.17 The $TRUNCATE and $NOTRUNCATE Metacommands

■ Action

$TRUNCATE enables truncation of all variable names to six characters. **$NOTRUNCATE** disables the default or a previous **$TRUNCATE** metacommand.

■ Syntax

$[NO]TRUNCATE

■ Remarks

The default is **$TRUNCATE**. If the **$STRICT** metacommand is set, any names greater than 6 characters will generate warning messages. This can make it easier to port your code to other systems.

If **$NOTRUNCATE** is set, the first 31 characters in a name are significant. Your operating system may also restrict the length of names.

Example

```
C       This program produces the following output:
C
C       74 Las Vegas St.
C       74 Las Vegas St.
C
C       Barry Floyd
C       3 Prospect Drive


        IMPLICIT character*20 (S)

$TRUNCATE
        StudentName =     'Enrique Pieras'
        StudentAddress = '74 Las Vegas St.'

        WRITE (*,100) StudentName, StudentAddress

$NOTRUNCATE
        StudentName =     'Barry Floyd'
        StudentAddress = '3 Prospect Drive'

        WRITE (*,100) StudentName, StudentAddress

 100    FORMAT(/1X,A20,/1X,A20)

        END
```

Appendixes

Appendix A
ASCII Character Codes

Dec	Oct	Hex	Chr	Dec	Oct	Hex	Chr
000	000	00H	NUL	032	040	20H	SP
001	001	01H	SOH	033	041	21H	!
002	002	02H	STX	034	042	22H	"
003	003	03H	ETX	035	043	23H	#
004	004	04H	EOT	036	044	24H	$
005	005	05H	ENQ	037	045	25H	%
006	006	06H	ACK	038	046	26H	&
007	007	07H	BEL	039	047	27H	'
008	010	08H	BS	040	050	28H	(
009	011	09H	HT	041	051	29H	)
010	012	0AH	LF	042	052	2AH	*
011	013	0BH	VT	043	053	2BH	+
012	014	0CH	FF	044	054	2CH	,
013	015	0DH	CR	045	055	2DH	-
014	016	0EH	SO	046	056	2EH	.
015	017	0FH	SI	047	057	2FH	/
016	020	10H	DLE	048	060	30H	0
017	021	11H	DC1	049	061	31H	1
018	022	12H	DC2	050	062	32H	2
019	023	13H	DC3	051	063	33H	3
020	024	14H	DC4	052	064	34H	4
021	025	15H	NAK	053	065	35H	5
022	026	16H	SYN	054	066	36H	6
023	027	17H	ETB	055	067	37H	7
024	030	18H	CAN	056	070	38H	8
025	031	19H	EM	057	071	39H	9
026	032	1AH	SUB	058	072	3AH	:
027	033	1BH	ESC	059	073	3BH	;
028	034	1CH	FS	060	074	3CH	<
029	035	1DH	GS	061	075	3DH	=
030	036	1EH	RS	062	076	3EH	>
031	037	1FH	US	063	077	3FH	?

Dec = Decimal, Oct = Octal, Hex = Hexadecimal(H), Chr = Character, LF = Line Feed, FF = Form Feed, CR = Carriage Return, DEL = Rubout

Appendix A *(continued)*

Dec	Oct	Hex	Chr	Dec	Oct	Hex	Chr
064	100	40H	@	096	140	60H	`
065	101	41H	A	097	141	61H	a
066	102	42H	B	098	142	62H	b
067	103	43H	C	099	143	63H	c
068	104	44H	D	100	144	64H	d
069	105	45H	E	101	145	65H	e
070	106	46H	F	102	146	66H	f
071	107	47H	G	103	147	67H	g
072	110	48H	H	104	150	68H	h
073	111	49H	I	105	151	69H	i
074	112	4AH	J	106	152	6AH	j
075	113	4BH	K	107	153	6BH	k
076	114	4CH	L	108	154	6CH	l
077	115	4DH	M	109	155	6DH	m
078	116	4EH	N	110	156	6EH	n
079	117	4FH	O	111	157	6FH	o
080	120	50H	P	112	160	70H	p
081	121	51H	Q	113	161	71H	q
082	122	52H	R	114	162	72H	r
083	123	53H	S	115	163	73H	s
084	124	54H	T	116	164	74H	t
085	125	55H	U	117	165	75H	u
086	126	56H	V	118	166	76H	v
087	127	57H	W	119	167	77H	w
088	130	58H	X	120	170	78H	x
089	131	59H	Y	121	171	79H	y
090	132	5AH	Z	122	172	7AH	z
091	133	5BH	[	123	173	7BH	{
092	134	5CH	\	124	174	7CH	|
093	135	5DH	]	125	175	7DH	}
094	136	5EH	^	126	176	7EH	~
095	137	5FH	_	127	177	7FH	DEL

Dec = Decimal, Oct = Octal, Hex = Hexadecimal(H), Chr = Character, LF = Line Feed, FF = Form Feed, CR = Carriage Return, DEL = Rubout

Appendix B
Intrinsic Functions

This appendix gives an alphabetical listing of all of the intrinsic functions in Microsoft FORTRAN. The following list shows the abbreviations used in this appendix for different data types.

Abbreviation	Data Type
gen	More than one possible argument type; see "Argument Type" column
int	**INTEGER**, INTEGER*1, INTEGER*2, or INTEGER*4
int1	INTEGER*1
int2	INTEGER*2
int4	**INTEGER***4
real	**REAL**, REAL*4, **DOUBLE PRECISION**, or REAL*8
real4	**REAL***4
dbl	**DOUBLE PRECISION**
log	**LOGICAL**, LOGICAL*1, LOGICAL*2, or LOGICAL*4
log1	LOGICAL*1
log2	LOGICAL*2
log4	**LOGICAL***4
cmp	**COMPLEX**, COMPLEX*8, or COMPLEX*16
cmp8	**COMPLEX***8
cmp16	COMPLEX*16
char	**CHARACTER** [*n]

Table B.1 gives an alphabetical listing of the intrinsic functions in Microsoft FORTRAN.

Table B.1
Intrinsic Functions

Name	Definition	Argument Type	Function Type
ABS(*gen*)	Absolute value	int, real, or cmp	Same as argument type unless argument is cmp[a]
ACOS(*real*)	Arc cosine	real	Same as argument
AIMAG(*cmp8*)	Imaginary part of cmp8 number	cmp8	real4
AINT(*real*)	Truncate	real	Same as argument
ALOG(*real4*)	Natural logarithm	real4	real4
ALOG10(*real4*)	Common logarithm	real4	real4
AMAX0(*intA*,*intB*[,*intC*]...)	Maximum	int	real4
AMAX1(*real4A*,*real4B*[,*real4C*]...)	Maximum	real4	real4
AMIN0(*intA*,*intB*[,*intC*]...)	Minimum	int	real4
AMIN1(*real4A*,*real4B*[,*real4C*]...)	Minimum	real4	real4
AMOD(*real4A*,*real4B*)	Remainder	real4	real4
ANINT(*real*)	Round	real	Same as argument
ASIN(*real*)	Arc sine	real	Same as argument
ATAN(*real*)	Arc tangent	real	Same as argument
ATAN2(*realA*,*realB*)	Arc tangent (*realA*/*realB*)	real	Same as argument
BTEST(*intA*,*intB*)	Bit test	int	log

Table B.1 *(continued)*

Name	Definition	Argument Type	Function Type
CABS(*cmp*)	Absolute value	cmp	real[a]
CCOS(*cmp8*)	Cosine	cmp8	cmp8
CDABS(*cmp16*)	Absolute value	cmp16	dbl
CDCOS(*cmp16*)	Cosine	cmp16	cmp16
CDEXP(*cmp16*)	Exponent	cmp16	cmp16
CDLOG(*cmp16*)	Natural logarithm	cmp16	cmp16
CDSIN(*cmp16*)	Sine	cmp16	cmp16
CDSQRT(*cmp16*)	Square root	cmp16	cmp16
CEXP(*cmp8*)	Exponent	cmp8	cmp8
CHAR(*int*)	Data-type conversion	int	char
CLOG(*cmp8*)	Natural logarithm	cmp8	cmp8
CMPLX(*genA*[,*genB*])	Data-type conversion	int, real, or cmp	cmp8
CONJG(*cmp8*)	Conjugate of cmp8 number	cmp8	cmp8
COS(*gen*)	Cosine	real or cmp	Same as argument
COSH(*real*)	Hyperbolic cosine	real	Same as argument
COTAN(*real*)	Cotangent	real	Same as argument
CSIN(*cmp8*)	Sine	cmp8	cmp8
CSQRT(*cmp8*)	Square root	cmp8	cmp8
DABS(*dbl*)	Absolute value	dbl	dbl
DACOS(*dbl*)	Arc cosine	dbl	dbl
DASIN(*dbl*)	Arc sine	dbl	dbl

Table B.1 *(continued)*

Name	Definition	Argument Type	Function Type
DATAN(*dbl*)	Arc tangent	dbl	dbl
DATAN2(*dblA*,*dblB*)	Arc tangent (*dblA*/*dblB*)	dbl	dbl
DBLE(*gen*)	Data-type conversion	int, real, or cmp	dbl
DCMPLX(*genA*[,*genB*])	Data-type conversion	int, real, or cmp	cmp16
DCONJG(*cmp16*)	Conjugate of cmp16 number	cmp16	cmp16
DCOS(*dbl*)	Cosine	dbl	dbl
DCOSH(*dbl*)	Hyperbolic cosine	dbl	dbl
DCOTAN(*dbl*)	Cotangent	dbl	dbl
DDIM(*dblA*,*dblB*)	Positive difference	dbl	dbl
DEXP(*dbl*)	Exponent	dbl	dbl
DIM(*genA*,*genB*)	Positive difference	int or real	Same as argument
DIMAG(*cmp16*)	Imaginary part of cmp16 number	cmp16	dbl
DINT(*dbl*)	Truncate	dbl	dbl
DLOG(*dbl*)	Natural logarithm	dbl	dbl
DLOG10(*dbl*)	Common logarithm	dbl	dbl
DMAX1(*dblA*,*dblB*[,*dblC*]...)	Maximum	dbl	dbl
DMIN1(*dblA*,*dblB*[,*dblC*]...)	Minimum	dbl	dbl
DMOD(*dblA*,*dblB*)	Remainder	dbl	dbl
DNINT(*dbl*)	Round	dbl	dbl
DPROD(*real4A*,*real4B*)	Double-precision product	real4	dbl

Table B.1 *(continued)*

Name	Definition	Argument Type	Function Type
DREAL(*cmp16*)	Data-type conversion	cmp16	dbl
DSIGN(*dblA*,*dblB*)	Sign transfer	dbl	dbl
DSIN(*dbl*)	Sine	dbl	dbl
DSINH(*dbl*)	Hyperbolic sine	dbl	dbl
DSQRT(*dbl*)	Square root	dbl	dbl
DTAN(*dbl*)	Tangent	dbl	dbl
DTANH(*dbl*)	Hyperbolic tangent	dbl	dbl
EOF(*gen*)	End-of-file	int	log
EXP(*gen*)	Exponent	real or cmp	Same as argument
FLOAT(*int*)	Data-type conversion	int	real4
HFIX(*gen*)	Data-type conversion	int, real, or cmp	int2
IABS(*int*)	Absolute value	int	int
IAND(*intA*,*intB*)	Logical product	int	Same as argument
IBCHNG(*intA*,*intB*)	Bit change	int	Same as argument
IBCLR(*intA*,*intB*)	Bit clear	int	Same as argument
IBSET(*intA*,*intB*)	Bit set	int	Same as argument
ICHAR(*char*)	Data-type conversion	char	int
IDIM(*intA*,*intB*)	Positive difference	int	int
IDINT(*dbl*)	Data-type conversion	dbl	int
IDNINT(*dbl*)	Round	dbl	int

Table B.1 *(continued)*

Name	Definition	Argument Type	Function Type
IEOR(*intA*,*intB*)	Exclusive or	int	Same as argument
IFIX(*real4*)	Data-type conversion	real4	int
IMAG(*cmp*)	Imaginary part of cmp number	cmp	real4
INDEX(*charA*,*charB*)	Location of substring *charB* in string *charA*	char	int
INT(*gen*)	Data-type conversion	int, real, or cmp	int
INT1(*gen*)	Data-type conversion	int, real, or cmp	int1
INT2(*gen*)	Data-type conversion	int, real, or cmp	int2
INT4(*gen*)	Data-type conversion	int, real, or cmp	int4
INTC(*gen*)	Data-type conversion	int, real, or cmp	INTEGER(C)
IOR(*intA*,*intB*)	Inclusive or	int	Same as argument
ISHA(*intA*,*intB*)	Arithmetic shift	int	Same as argument
ISHC(*intA*,*intB*)	Rotate	int	Same as argument
ISHFT(*intA*,*intB*)	Logical shift	int	Same as argument
ISHL(*intA*,*intB*)	Logical shift	int	Same as argument
ISIGN(*intA*,*intB*)	Sign transfer	int	int
JFIX(*gen*)	Data-type conversion	int, real, or cmp	int4
LEN(*char*)	Length of string	char	int

Table B.1 *(continued)*

Name	Definition	Argument Type	Function Type
LGE(*charA*,*charB*)	*charA* >= *charB*	char	log
LGT(*charA*,*charB*)	*charA* > *charB*	char	log
LLE(*charA*,*charB*)	*charA* <= *charB*	char	log
LLT(*charA*,*charB*)	*charA* < *charB*	char	log
LOC(*gen*)	Address	Any	int2 or int4
LOCFAR(*gen*)	Segmented address	Any	int4
LOCNEAR(*gen*)	Unsegmented address	Any	int2
LOG(*gen*)	Natural logarithm	real or cmp	Same as argument
LOG10(*real*)	Common logarithm	real	Same as argument
MAX(*genA*,*genB*‖,*genC*‖...)	Maximum	int or real	Same as argument
MAX0(*intA*,*intB*‖,*intC*‖...)	Maximum	int	int
MAX1(*real4A*,*real4B*‖,*real4C*‖...)	Maximum	real4	int
MIN(*genA*,*genB*‖,*genC*‖...)	Minimum	int or real	Same as argument
MIN0(*intA*,*intB*‖,*intC*‖...)	Minimum	int	int
MIN1(*real4A*,*real4B*‖,*real4C*‖...)	Minimum	real4	int
MOD(*genA*,*genB*)	Remainder	int or real	Same as argument
NINT(*real*)	Round	real	int
NOT(*intA*)	Logical complement	int	Same as argument
REAL(*gen*)	Data-type conversion	int, real, or cmp	real4
SIGN(*genA*,*genB*)	Sign transfer	int or real	Same as argument
SIN(*gen*)	Sine	real or cmp	Same as argument

Table B.1 *(continued)*

Name	Definition	Argument Type	Function Type
SINH(*real*)	Hyperbolic sine	real	Same as argument
SNGL(*dbl*)	Data-type conversion	dbl	real4
SQRT(*gen*)	Square root	real or cmp	Same as argument
TAN(*real*)	Tangent	real	Same as argument
TANH(*real*)	Hyperbolic tangent	real	Same as argument

[a] If argument is **COMPLEX*8**, function is **REAL*4**. If argument is COMPLEX*16, function is DOUBLE PRECISION.

Appendix C
Additional Procedures

C.1 Introduction

Microsoft FORTRAN contains some additional procedures that control and access the system time and date, and get and reset run-time-error code information. The following sections describe these procedures.

C.2 Time and Date Procedures

The functions **SETTIM** and **SETDAT**, and the subroutines **GETTIM** and **GETDAT**, allow you to use the system time and date in your programs. **SETTIM** and **SETDAT** set the system time and date; **GETTIM** and **GETDAT** return the time and date. Table C.1 summarizes the time and date procedures.

Table C.1

Time and Date Procedures

Name	Definition	Argument Type	Function Type
GETTIM(*ihr,imin,isec,i100th*)	Gets system time	**INTEGER*2**	
SETTIM(*ihr,imin,isec,i100th*)	Sets system time	**INTEGER*2**	LOGICAL
GETDAT(*iyr,imon,iday*)	Gets system date	**INTEGER*2**	
SETDAT(*iyr,imon,iday*)	Sets system date	**INTEGER*2**	LOGICAL

The arguments are defined as follows:

Argument	Definition
ihr	Hour (0 – 23)
imin	Minute (0 – 59)
isec	Second (0 – 59)
i100th	Hundredth of a second (0 – 99)
iyr	Year (*xxxx* AD)

imon	Month (1–12)
iday	Day of the month (1–31)

Actual arguments used in calling **GETTIM** and **GETDAT** must be **INTEGER*2** variables or array elements. Because these subroutines redefine the values of their arguments, other kinds of expressions are prohibited.

Actual arguments of the functions **SETTIM** and **SETDAT** can be any legal **INTEGER*2** expression. **SETTIM** and **SETDAT** return **.TRUE.** if successful (that is, the system time or date is changed), or **.FALSE.** if no change is made.

Refer to your operating-system documentation for the range of permitted dates.

Examples

The following program sets the date and time, then prints them on the screen:

```
c Warning: this program will reset your
c system date and time.

$STORAGE:2
      CHARACTER*12 CDATE, CTIME
      LOGICAL SETDAT, SETTIM
      DATA CDATE /'The date is '/
      DATA CTIME /'The time is '/
      IF (.NOT. (SETDAT(1986,7,4)))
     + WRITE (*,*) 'SETDAT failed'
c sets the date to July 4th, 1986.
      IF (.NOT. (SETTIM (0,0,0,0)))
     + WRITE (*,*) 'SETTIM failed'
c sets the time to 00:00:00.00 (midnight)
      CALL GETDAT(IYR,IMON,IDAY)
c gets the date from the system clock
      CALL GETTIM(IHR,IMIN,ISEC,I100TH)
c gets the time of day from the system clock
     + CDATE,IMON,IDAY,IYR
c writes out the date
      WRITE (*, '(1X,A,I2.2,1H:,I2.2,1H:
     + ,I2.2,1H.,I2.2)')
     + CTIME,IHR,IMIN,ISEC,I100TH
c writes out the time in the format xx:xx:xx.xx
      END
```

C.3 Run-Time-Error Procedures

The **IGETER** function and the **ICLRER** subroutine are included for compatability with previous versions of FORTRAN. Their functionality is provided in the current version by the **IOSTAT=** option. (See Section 4.3.10, "Error and End-of-File Handling," for more information about **IOSTAT=**.)

IGETER is called after an I/O operation that includes the **ERR=** or **IOSTAT=** options. It returns the following values:

Return Value	Description
0	No error occurred.
Negative value	An end-of-file condition occurred, but no other error occurred.
Positive value	An error occurred. The return value is the error number.

The **IGETER** calling interface has the following form:

```
INTEGER*2 FUNCTION IGETER(IUNIT)
INTEGER*2 IUNIT
.
.
.
END
```

ICLRER resets the FORTRAN run-time-error code information after an error has been encountered and handled through **ERR=** and **IOSTAT=**. The **ICLRER** calling interface has the following form:

```
SUBROUTINE ICLRER(IUNIT)
INTEGER*2 IUNIT
.
.
.
END
```

Glossary

The definitions in this glossary are intended primarily for use with this manual and the *Microsoft FORTRAN Compiler User's Guide*. Neither individual definitions nor the list of terms is comprehensive.

8087 and 80287 coprocessors

Intel® hardware products that provide very fast and precise number processing.

Actual argument

The specific item (such as a variable, array, or expression) passed to a subroutine or function at a specific calling location.

Alphanumeric

A letter or a number.

Argument

A value passed to and from functions and subroutines.

Array declarator

The specifier *array*([[*lower*:]]*upper*).

Associated

Referring to the same actual memory location.

Attribute

A keyword that specifies additional information about a variable, variable type, subprogram, or subprogram formal argument.

Base name

The portion of the file name that precedes the file-name extension. For example, `samp` is the base name of the file `samp.for`.

Binary

Base-2 notation.

C string

A character constant followed by the character C. The character constant is then interpreted as a C-language constant.

Column-major order

The order in which array elements are stored: the leftmost subscript is incremented first when the array is mapped into contiguous memory addresses.

Compiland

A file containing ASCII text to be compiled by the Microsoft FORTRAN Compiler. A compiland is also called a source file.

Compile time

The time during which the compiler is executing, compiling a Microsoft FORTRAN source file, and creating a relocatable object file.

Compiler

A program that translates FORTRAN programs into code understood by the computer.

Complex number

A number with a real and an imaginary part.

Constant folding

The process of evaluating expressions that contain only constants, and then substituting the calculated constant for the expression. Constant folding is performed by the compiler during optimization. The expression `9*3.1`, for example, becomes `27.9`.

Dimension declarator

The specifier [*lower*:]*upper*; *upper* minus *lower* plus 1 equals the number of dimensions in an array.

Domain

The range of a function's valid input values. For example, in the expression `y=f(x)`, the domain of the function `f(x)` is the set of all values of `x` for which `f(x)` is defined. A "DOMAIN" error message is returned when an argument to a function is outside the domain of the function.

Double-precision real number

A real number that is allocated 8 bytes of memory.

Dummy argument

A formal argument.

Excess-127, Excess-64

A type of notation in which the specified constant (127 or 64) is added to every number.

Executable program

A file containing executable program code. When the name of the file is entered at the system prompt, the instructions in the program are performed.

External reference

A variable or routine in a given module that is referred to by a routine in another module.

Far call

An address that specifies the segment as well as the offset.

FL

A command used by Microsoft FORTRAN to compile and link programs.

Formal argument

The name by which a specific argument is known within a function or subroutine.

Hexadecimal

Base-16 notation.

High-order bit

The highest-numbered bit; the bit farthest to the left. It is also called the most-significant bit.

Huge model

A memory model that allows for more than one segment of code and more than one segment of data, and that allows individual data items to span more than one segment.

IEEE

Institute of Electrical and Electronics Engineers, Inc.

In-line code

Code that is in the main program, as opposed to code that is in a subroutine called by the main program. Using in-line code is faster, but it makes programs larger.

Input/output list (I/O)

A list of items to input or output. **PRINT**, **READ**, or **WRITE** statements can specify an I/O list.

Keyword

A word with a special, predefined meaning for the compiler.

Large model

A memory model that allows for more than one segment of code and more than one segment of data.

Large-model compiler

A compiler that assumes a program has more than one segment of code and more than one segment of data.

Least-significant byte

The lowest-numbered byte; the first byte. It is also called the low-order byte.

Library

A file that stores related modules of compiled code. These modules are used by the linker to create executable program files.

Link time

The time during which the linker is executing, that is, linking relocatable object files and library files.

Linking

The process by which the linker loads modules into memory, computes addresses for routines and variables in relocatable modules, and then resolves all external references by searching the run-time library. After loading and linking, the linker saves the modules it has loaded into memory as a single executable file.

Long call

An address that specifies the segment as well as the offset. It is also referred to as the long address.

Low-order bit

The lowest-numbered bit; the bit farthest to the right. It is also called the least-significant bit.

Machine code

Instructions that a microprocessor can execute.

Mantissa

The decimal part of a base-10 logarithm.

Medium model

A memory model that allows for more than one segment of code and only one segment of data.

Memory map

A representation of where in memory the compiler expects to find certain types of information.

Most-significant byte

The highest-numbered byte; the last byte. It is also called the high-order byte.

NAN

An abbreviation that stands for "Not A Number." NANs are generated when the result of an operation cannot be represented in the IEEE format. For example, if you try to add two positive numbers whose sum is larger than the maximum value permitted by the compiler, the processor will return a NAN instead of the sum.

Near call

A call to a routine in the same segment. The address of the called routine is specified with an offset.

Object file

A file that contains relocatable machine code.

Offset

The number of bytes from the beginning of a segment to a particular byte in that segment.

Optimize

To reduce the size of the executable file by eliminating unnecessary instructions.

Pass

Individual readings of source code made by the compiler as it processes information. Each reading is called a pass.

PLOSS

Appears in an error message when the error caused a partial loss of accuracy in the significant digits of the result. For example, a PLOSS error on a single-precision result indicates that less than six decimal digits of the result are reliable.

Program unit

A main program, a subroutine, a function, or a block-data subprogram.

Relocatable

Not containing absolute addresses.

Run time

The time during which a previously compiled and linked program is executing.

Run-time library

A file containing the routines needed to implement certain functions of the Microsoft FORTRAN language. A library module usually corresponds to a feature or subfeature of the Microsoft FORTRAN language.

Segment

An area of memory, less than or equal to 64K long, containing code or data.

Short call

A call to a routine in the same segment. The address of the called routine is specified with only an offset. It is also referred to as the short address.

Sign extended

The sign bit of a number is propagated through all the higher-order bits. In this way, the sign is preserved when the number is written into a larger format.

Single-precision real number

A real number that is allocated 4 bytes of memory.

Small model

A memory model that allows for only one segment of code and only one segment of data.

Source file

A file containing the original ASCII text of a Microsoft FORTRAN program.

Stack

A dynamically shrinking and expanding area of memory in which data items are stored in consecutive order and removed on a last-in, first-out basis.

String

A character constant.

Terminal I/O

Any I/O done to a terminal device. Examples of a terminal device are the console, keyboard, and printer.

TLOSS

Appears in an error message when the error caused a total loss of accuracy in the significant digits of the result. For example, a TLOSS error on a single-precision result indicates that none of the six significant digits of the result are reliable.

Two's complement

A type of base-2 notation in which 1s and 0s are reversed (complemented), and 1 is added to the result.

Type coercion

The forcing of a variable to have a particular data type. For example, when integer values are used in expressions, if one operand of an expression containing the operators plus (+), minus (−), or multiplication (*) is of type **REAL**, the other operand is converted to a real number before the operation is performed.

Undefined variable

A variable that cannot be found, either in the routine being linked or, for an external reference, in a routine in another module.

Unresolved reference

A reference to a variable or a subprogram that cannot be found, either in the routine being linked or, for an external reference, in a routine in another module.

Index