Microsoft® FORTRAN

Reference

Microsoft®

Microsoft® FORTRAN

REFERENCE

VERSION 5.1

FOR MS® OS/2® AND MS-DOS®
OPERATING SYSTEMS

MICROSOFT CORPORATION

Printed and bound in the United States of America.

Document No. LN21013-0591

10 9 8 7 6 5 4 3 2 1

Table of Contents Overview

Table of Contents

Introduction

PART 1 Language Reference

PART 2 Compiling and Linking

Appendixes

Figures and Tables

Figures

Tables

Introduction

Microsoft® FORTRAN versions 5.1 and 5.0 improve on the already popular Microsoft FORTRAN programming language by adding several important new features. The language is now fully compatible with the Systems Application Architecture (SAA) FORTRAN extensions and many of the VAX® extensions. It supports new constructs such as compound data types (structures) and **SELECT CASE** decision-making blocks. New compiler directives allow advanced features like conditional compilation of specific pieces of program code. Also, the FL compiling and linking command has several new options and improvements to existing options.

This new version of Microsoft FORTRAN offers enhanced OS/2® systems support, including the use of dynamic-link libraries and multiple threads of execution. For complete flexibility, programs can be designed to run under DOS, OS/2, or both operating systems. In addition, an extensive graphics library lets data and figures become an integral part of any FORTRAN application.

This chapter introduces the *Microsoft FORTRAN Reference*, describes the document conventions used in the manual, and gives additional sources of information about FORTRAN.

For discussions of memory models, calling non-FORTRAN subroutines and functions from a Microsoft FORTRAN program (mixed-language programming), programming under OS/2, and the use of graphics, see *Microsoft FORTRAN Advanced Topics*. To find out how to use the Microsoft CodeView® Window-Oriented Debugger to debug your programs, see the *Microsoft CodeView and Utilities User's Guide.*

About This Manual

The *Microsoft FORTRAN Reference* defines the FORTRAN language as implemented by the Microsoft FORTRAN Optimizing Compiler, Version 5.0. 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 texts on FORTRAN, see "Books about FORTRAN Programming" at the end of this introduction.

Microsoft documentation uses the term "OS/2" to refer to the OS/2 systems—Microsoft Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" refers to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to the system.

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

Chapter 1 discusses the elements of the FORTRAN programming language. Chapter 2 explains the structure of FORTRAN programs. Chapter 3 gives the details of FORTRAN's input/output (I/O) system. Chapter 4 is a detailed description of all FORTRAN statements. Chapter 5 explains all of FORTRAN's intrinsic functions. Chapter 6 covers Microsoft FORTRAN's metacommands, and Chapter 7 discusses the FL command. The following list shows where to look for information on specific topics:

Topic	Location of Information
Characters, names, data types, attributes, and expressions in FORTRAN	Chapter 1, "Elements of FORTRAN"
Formatting lines in your source program; subroutines, functions, and arguments; and structuring your FORTRAN programs	Chapter 2, "Program Structure"
Input and output in FORTRAN	Chapter 3, "The Input/Output System"
FORTRAN statements, listed alphabetically	Chapter 4, "Statements"
FORTRAN intrinsic functions, listed alphabetically	Chapter 5, "Intrinsic Functions"
Compiler directives, called metacommands, listed alphabetically	Chapter 6, "Metacommands"
FL command	Chapter 7, "The FL Command"

Table of the American Standard Code for Information Interchange (ASCII) character set	Appendix A, "ASCII Character Codes"
Selected terms used in this documentation	"Glossary"

Document Conventions

This manual uses the following typographic conventions. (Note that, in most cases, blanks are not significant in FORTRAN).

Example of Convention	Description of Convention
Extensions to the ANSI standard language	Blue type indicates 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.
OUT.TXT, ANOVA.EXE, COPY, LINK, FL	Uppercase (capital) letters indicate file names and DOS-level commands. Uppercase is also used for command-line options (unless the application accepts only lowercase).

`C  Comment line`
`WRITE (*,*) 'Hello`
`+World'`
	This kind of type is used for program examples, program output, DOS-level commands, and error messages within the text. A capital C marks the beginning of a comment in sample programs. Continuation lines are indicated by a plus sign (+) in column 6.

AUTOMATIC, INTRINSIC, WRITE	Bold capital letters indicate language-specific keywords with special meaning to FORTRAN. Keywords are a required part of statement syntax, unless enclosed in double brackets as explained below. In programs you write, FORTRAN keywords are entered in all-uppercase (capital) letters, or any combination of uppercase and lowercase letters.

other keywords	Bold lowercase letters are used for 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.
Apostrophes: '' ''	In Microsoft FORTRAN, 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: `'`.
expression	Words in italics indicate placeholders for information that you must supply. A file name is an example of this kind of information. Italics are also occasionally used in the text for emphasis.
[[*optional item*]]	Items inside double square brackets are optional.
{*choice1* | *choice2*}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of the items unless all of the items are also enclosed in double square brackets.
Repeating elements...	Three dots following an item indicate that more items having the same form may be entered.

`CALL GetNum (i, *10)`
.
.
.
`SUBROUTINE GetNum (i, *)`
	A column of three dots indicates that part of the example has intentionally been omitted.

KEY NAMES	Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+C. A plus (+) indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key. The carriage-return key, sometimes marked with a bent arrow, is referred to as ENTER. The cursor movement ("arrow") keys on the numeric keypad are called DIRECTION keys. Individual DIRECTION keys are referred to by the direction of the arrow on the key top (LEFT, RIGHT, UP, DOWN) or the name on the key top (PGUP, PGDN). The key names used in this manual correspond to the name on the IBM Personal Computer keys. Other machines may use different names.
"defined term"	Quotation marks usually indicate a new term defined in the text.
Video Graphics Array (VGA)	Acronyms are usually spelled out the first time they are used.

Example

The following example shows how this manual's typographic 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.

Books about FORTRAN Programming

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

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

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

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

Ledgard, Henry F., and L. Chmura. *FORTRAN with Style*. Rochelle Park, NJ: Hayden, 1978.

Wagener, Jerrold L. *FORTRAN 77: Principles of Programming*. New York, NY: Wiley, 1980.

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

Requesting Assistance

If you need help or feel you have discovered a problem in the software, please provide the following information to help us locate the problem:

- The version of DOS you are running (use the DOS VER command)
- Your system configuration:
  1. Make and model of your computer
  2. Total memory and total free memory at compiler execution time (use the DOS CHKDSK command to obtain these values)
  3. Any other information you think might be useful
- The compiler command line used (or the link command line if the problem occurred during linking)
- Any object files or libraries you linked with if the problem occurred at link time

If your program is large, please try to reduce it to the smallest possible program that still produces the problem.

Use the Product Assistance Request form at the back of this manual to send this information to Microsoft.

If you have comments or suggestions regarding any of the manuals accompanying this product, please indicate them on the Document Feedback card at the back of this manual.

If you are not already a registered Microsoft FORTRAN owner, you should fill out and return the Registration Card. This enables Microsoft to keep you informed of updates and other information.

PART 1

Language Reference

PART 1

Language Reference

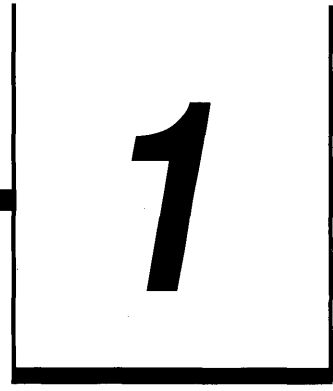
This section describes Version 5.0 of the Microsoft FORTRAN language—both its ANSI-standard features and the special Microsoft extensions, which are presented in blue type. The first three chapters introduce the language by reviewing the elements of FORTRAN (Chapter 1), giving an overview of FORTRAN program structure (Chapter 2), and discussing the FORTRAN input/output system (Chapter 3).

The remaining two chapters provide the information you will need to put these features to work. Chapter 4 is an alphabetical statement reference, with information on syntax and usage for each Microsoft FORTRAN statement. Chapter 5 provides information on intrinsic functions and additional procedures supplied with this package.

CHAPTERS

CHAPTER 1

Elements of FORTRAN



This chapter explains the building blocks of FORTRAN programs: special characters, the scope and naming of FORTRAN identifiers, the data types that are available, and the rules that govern their use. The basic arithmetic and logical functions available are also covered.

1.1 Characters

FORTRAN source files can contain any printable characters in the ASCII character set. When character constants or character variables are logically compared, the collating sequence for the FORTRAN character set is the ASCII sequence. 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).

  The Microsoft FORTRAN Compiler interprets lowercase letters as uppercase letters in all contexts except character constants and 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 a part of the FORTRAN statements listed in Chapter 4, "Statements," are not case sensitive unless the **$STRICT** metacommand is specified. For more information, see Chapter 6, "Metacommands."

  When using 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'**.

- The 10 digits (0 through 9). Digits can be included in user-defined names; however, a digit may not be the first character.
- All other printable characters in the ASCII character set, the blank character, and the TAB character.

In Microsoft FORTRAN, the dollar sign ($) and the underscore (_) can be included in user-defined names. The underscore, however, cannot be the first character in a name.

The blank character has no significance in a FORTRAN source file (except as listed below) so you can insert 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 2.1, "Lines," for an explanation of initial lines).

The tab character's interpretation depends on which column it 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 a single blank, unless it is in a character or Hollerith constant (described in Section 3.7.1.2). A tab character in a character or Hollerith constant is interpreted as a tab character.

1.2 Names

All variables, arrays, functions, programs, and subprograms are identified by names. A name is a sequence of alphanumeric characters and must follow these guidelines:

- The first character in a name must be alphabetic; the rest of the characters must be alphanumeric. Microsoft FORTRAN allows the dollar sign ($) as an alphabetic character that follows Z in the **IMPLICIT** collating sequence for names. The underscore may also appear in names, but it may not be the first character.
- Blanks are ignored. Variable names like `low volt age` and `lowvoltage` are identical to the compiler.

- Names may be up to 31 characters long; all characters are significant. Only the first six alphanumeric characters are significant and the rest are ignored. (This limitation does not apply to Microsoft FORTRAN unless the **$STRICT** or **$TRUNCATE** metacommand is in effect.) Blank characters do not count. For example, the names `delicate` and `d eli cat e` are both interpreted as `delica` if **$TRUNCATE** or **$STRICT** is set.
- Your operating system or linker may impose other limits on name lengths.

FORTRAN keywords are not reserved 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 keyword names for variables, however, makes programs harder to read and understand. For readability, and to reduce the possibility of hard-to-find bugs, programmers should avoid using names that look like parts of FORTRAN statements.

Another type of error occurs when a typographical error causes the compiler to interpret a FORTRAN keyword as part of a variable name. Consider the following two statements:

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

The first statement is the beginning of a **DO** loop. The second statement assigns the value `1.20` to a variable named `DO5INC`. The only difference between the two statements is that the first contains a comma and the second contains a period. The compiler cannot catch this type of error. The terminating line for the **DO** loop (a subsequent line labeled `5`) is no different from any other line in the program, so the compiler has no way to recognize that an intended **DO** statement is missing.

Because the compiler reserves as external names the following three predefined names, they cannot be used to name a program:

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 2.7, "Main Program," for more information.)
2. **COMMQQ**, which is the system name for blank common blocks.
3. **BLKDQQ**, which is the default system name for block-data subprograms.

Also, all names beginning with two underscore characters (`_ _`) or ending with QQ, such as `__main` or `MAINQQ`, are reserved as external names by the compiler. If you must use a name beginning with two underscore characters or ending with QQ, use the **ALIAS** attribute to provide an acceptable external name for the variable.

1.2.1 Global and Local Names

There are two basic types of names: global and local.

Global names are recognized anywhere in a program, so they can have only one global definition anywhere within that program. All subroutine, function, program, and common-block names are global. For example, if you use a subroutine named `Sort` in one program, you cannot also have a function named `Sort` in that program.

You can, however, use `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 and a local name in the same program. This is permitted because common-block names are always enclosed in slashes, distinguishing them 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 have meaning only in a single program unit. In another program unit of the same program, the same name can be used again to refer to the same program object or to a different object.

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

Arguments to statement functions are a special case of local names. These arguments have meaning 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 the statement function arguments with the same name. See the Statement Function entry in Section 4.2 for more information.

1.2.2 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 **IMPLICIT NONE** has been declared, an error message is generated at the first use of any variable that has not been explicitly typed. If the **$DECLARE** metacommand is in effect, a warning message is generated at the first use of any variable that has not been given a value in a **DATA**, **PARAMETER**, or assignment statement. The following list explains how undeclared names are classified:

Use of Name	Classification
As a variable or in a function reference	The first letter of a variable or function determines its type. By default, variable names starting with I, J, K, L, M, or N (uppercase or lowercase) are given the type **INTEGER**, while variable 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).
As the target of a **CALL** statement	The compiler assumes the name is the name of a subroutine. A subroutine does not return a value through its name, so no typing occurs.

1.3 Data Types

The six basic data types in FORTRAN are described in this section. They are

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

The data type of a variable, array, symbolic constant, or function can be declared in a specification statement. If its type is not declared, the compiler determines a name's type by its first letter (as described in Section 1.2.2, "Undeclared Names"). A type statement can also dimension an array variable. In Microsoft FORTRAN a type statement can initialize variables and arrays. See Chapter 4, "Statements," for detailed descriptions of type statements.

The following sections describe each data type individually. Memory requirements are shown in Table 1.1.

Table 1.1 Memory Requirements

Type	Bytes	Notes
INTEGER	2 or 4	Defaults to 4 bytes. The setting of the **$STORAGE** metacommand determines the size of **INTEGER** and **LOGICAL** values.
INTEGER*1	1	
INTEGER*2	2	
INTEGER*4	4	
REAL	4	Same as **REAL*4**.
REAL*4	4	
DOUBLE PRECISION	8	Same as **REAL*8**.
REAL*8	8	
COMPLEX	8	Same as **COMPLEX*8**.
COMPLEX*8	8	
DOUBLE COMPLEX	16	Same as **COMPLEX*16**.
COMPLEX*16	16	
LOGICAL	2 or 4	Defaults to 4 bytes. The setting of the **$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.
RECORD	Size of structure type	Maximum 65,535 bytes; affected by **$PACK** metacommand.

1.3.1 Integer Data Types

An integer value is an exact representation of the corresponding integer. Table 1.2 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 two bytes long, by default. However, a constant outside the **INTEGER*2** range is given 4 bytes of storage.

Table 1.2 Integers

Data Type	Bytes	Range
INTEGER*1	1	–128 to 127
INTEGER*2	2	–32,768 to 32,767
INTEGER*4	4	–2,147,483,648 to 2,147,483,647
INTEGER	2 or 4	Depends on setting of **$STORAGE**

Although an **INTEGER*4** constant or variable can take any value in its full 32-bit range, it cannot be assigned its smallest negative value (–2,147,483,648) when that value is expressed as a numeric constant. For example, the statement `varname = -2147483648` causes a compiler overflow error. If you wish to give a four-byte integer this value, the right side of the assignment statement must be a mathematical expression.

Syntax

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

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

The optional *sign* is a 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 numbers greater than 9. For base 36, for example, A represents 10, B represents 11, C represents 12, and so on, through Z, which represents 35. The case of the letters is not significant.

Example

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

```
I = 2#1111001111001111001111
m = 7#45644664
J = +8#17171717
K = #3CF3CF
n = +17#2DE110
L = 3994575
index = 36#2DM8F
```

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 that is other than 10, the compiler reads out-of-range numbers up to 2^{32}. They are interpreted as the negative numbers with the corresponding internal representation. For example, `16#FFFFFFFF` results in an arithmetic value of –1. If the **$DEBUG** metacommand is in effect, a compile-time range error occurs instead.

1.3.2 The Single-Precision IEEE Real Data Type

A single-precision real value (**REAL** or **REAL*4**) is usually 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. You can specify more than six digits, but only the first six 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

A real constant has the following form:

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

Parameter	Value
sign	A sign (+ or –).
integer	An integer part, consisting of one or more decimal digits. 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 10 raised to the value *exponent*.

Example

The following real constants all represent the same real number (1.23):

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

1.3.3 The Double-Precision IEEE Real Data Type

A double-precision real value (**DOUBLE PRECISION** or **REAL*8**) is usually an approximation of the real number desired, and occupies 8 bytes of memory. The precision is between 15 and 16 decimal digits. You can specify more digits, but only the first 15 are significant. The range of double-precision real values includes the number 0 and the negative numbers from approximately –1.797693134862316D+308 to –2.225073858507201D–308. 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.

Example

The following double-precision real constants all represent fifty-two one-thousandths:

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

1.3.4 Complex Data Types

The **COMPLEX** or **COMPLEX*8** data type is an ordered pair of single-precision real numbers. The **DOUBLE COMPLEX** or **COMPLEX*16** data type is an ordered pair of double-precision real numbers. The first number in the pair is the real part of a complex number, and the second number in the pair is 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 **DOUBLE COMPLEX** or **COMPLEX*16** number are **REAL*8** numbers, so **DOUBLE COMPLEX** and **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	A real number or an integer, representing the real part.
imag	A real number or an integer, representing the imaginary part.

For example, the ordered pair (7,3.2) represents the complex number 7.0+3.2i. The ordered pair –(–.11E2,#5F) represents the complex number 11.0–95.0i.

1.3.5 Logical Data Types

The logical data type consists of two logical values, **.TRUE.** and **.FALSE.** A **LOGICAL** variable occupies 2 or 4 bytes of memory, depending on the setting of the **$STORAGE** metacommand. The default is 4 bytes. The value (true or false) of a logical variable is unaffected by the **$STORAGE** metacommand.

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 and 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.

1.3.6 The Character Data Type

Character variables occupy one byte of memory for each character and are assigned to contiguous bytes, independent of word boundaries. However, when character and noncharacter variables are allocated in the same common block, the compiler places noncharacter variables that follow character variables on word boundaries. See the **COMMON** entry in Section 4.2, "Statement Directory," for more information on character variables in common blocks.

The length of a character variable, character array element, structure element, character function, or character constant with a symbolic name must be between 1 and 32,767. The length can be specified by the following:

- An integer constant in the range 1– 32,767
- An expression (in parentheses) that evaluates to an integer in the range 1– 32,767
- An asterisk, which indicates the length of the string may vary

Examples

The following examples show proper usage:

```
CHARACTER*32 string
CHARACTER string*32
CHARACTER string*(const*5)
CHARACTER string*(*)
CHARACTER*(*) string
```

See the **CHARACTER** entry in Section 4.2, "Statement Directory," for more information.

A character constant is a sequence of one or more of the printable ASCII characters enclosed in a pair of apostrophes (') or quotes ("). The apostrophes or quotes that delimit the string are not stored with the string. To represent an apostrophe within a string delimited by apostrophes, specify two consecutive apostrophes with no blanks between them. To represent a quotation mark within a string delimited by quotation marks, specify two consecutive quotation marks with no blanks between them.

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

Blank characters and tab characters are permitted in character constants and are significant. The case of alphabetic characters is also significant. You can use C strings (as described in Section 1.3.6.1) to define strings with nonprintable characters, or to specify the null string.

The length of a character constant is equal to the number of characters between the apostrophes or quotation marks. (A pair of apostrophes in a string delimited by apostrophes counts as a single character. A pair of quotation marks in a string delimited by quotation marks counts as a single character.)

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 as one"`	`"Double" quotes count as one`

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 the lines. Consider the following FORTRAN statement:

```
C     Sample FORTRAN continuation line
      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 `Particles` ends in column 58, leaving 14 blanks to be added to the string.

1.3.6.1 C Strings

String values in the C language are terminated with null characters (**CHAR(0)**), and may contain nonprintable characters (such as newline and backspace). Nonprintable characters are specified using the backslash as an escape character, followed by a single character indicating the nonprintable character desired. This type of string is 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 1.3 shows the valid escape sequences. If a string contains an escape sequence that isn't in this table (such as \z), the backslash is ignored.

Table 1.3 C String Escape Sequences

Sequence	Character
\a	Bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\x*hh*	Hexadecimal bit pattern
\'	Single quote[1]
\"	Double quote
\\	Backslash
ooo	Octal bit pattern

[1] In FORTRAN you must enter \'' to indicate this escape sequence.

A C string must also be a valid FORTRAN string. Therefore, if the string is delimited by apostrophes, all apostrophes in the string itself must be represented by double apostrophes. The escape sequence `\'a` causes a compiler syntax error because FORTRAN interprets the quotation mark as the end of a string.

The correct form is `\''a`. (If the string is delimited with quotation marks, an apostrophe may be entered as a single character.) C strings and ordinary strings differ only in how you specify the value of the string. The compiler treates them identically.

The sequences *ooo* and \x*hh* allow any ASCII character 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. 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.

1.3.6.2 Character Substrings

Character substrings are used to access a contiguous part of a character variable.

Syntax

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

Parameter	Description
variable	A character variable or a character array element.
first	An integer expression or 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 integer expression or 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.

The length of the substring is *last* – *first* + 1. For example, if a 10-byte character variable `name` contains the string `'Jane Doe '`, the following is true:

Character Variable	Equivalent String
`name(:5)`	`'Jane '`
`name(6:)`	`'Doe '`
`name(3:7)`	`'ne Do'`
`name(:)`	`'Jane Doe '`

Note that `name(:)` is equivalent to `name`.

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

```
1 <= first <= last <= length
```

That is, both *first* and *last* must be greater than zero; *last* must be greater than or equal to *first*; and neither *first* nor *last* can be greater than *length*.

For the 10-byte character variable `name`, the following substring specifications are illegal:

```
name(0:4)
name(6:5)
name(11:12)
name(0:)
name(:11)
```

If the **$DEBUG** metacommand is in effect, a run-time error occurs if these relationships are not true. If **$DEBUG** is not in effect, 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
```

Output

```
nopqrstuvwxyzabcdefghijklm
```

The use of a noninteger expression for the *first* and *last* parameters causes an error. (This limitation does not apply to Microsoft FORTRAN unless the **$STRICT** metacommand is in effect.)

1.4 Records

A record is a "structure" variable. A structure is a user-defined compound data type that consists of variable type definitions and unions of maps as explained below. Each item within a structure is called a "structure element."

Example

The following example is a structure definition for a variable type that contains employee data:

```
STRUCTURE /employee_data/
   CHARACTER*25  last_name
   CHARACTER*15  first_name
   CHARACTER*20  street_name
   INTEGER*2     street_number
   INTEGER*2     apt_number
   CHARACTER*20  city
   CHARACTER*2   state
   INTEGER*4     zip
   INTEGER*4     telephone
   INTEGER*2     date_of_birth
   INTEGER*2     date_of_hire
   INTEGER*2     social_security(3)
   LOGICAL*2     married
   INTEGER*2     dependents
END STRUCTURE
```

A structure specification is not a variable, but a variable type. Structure variables are defined with the **RECORD** statement. In a company with 200 employees, the following statement would define an array of structure variables to hold employee data, using the previous structure definition:

```
RECORD /employee_data/ employees(200)
```

Structure elements are referenced by specifying the sequence of elements needed to reach the desired element. The elements are separated by a period. For example, the number of dependents of the ninety-ninth employee is specified by `employees(99).dependents`. The first letter of the name of the state where this employee lives is specified by `employees(99).state(1:1)`. Note that the data type of a structure reference is the type of the element referenced. In the two examples given here, the first reference is of type **INTEGER*2**. The second reference is of type **CHARACTER*2**. Any intervening references are for identification only; they do not affect the data type.

Because periods are used to delimit structure elements, an element may not have the name of a relational or logical operator (**NOT**, **AND**, **OR**, **GE**, **EQ**, etc.). If the name of a relational or logical operator is used, the compiler will try to interpret the name as a relational operator.

Example

A structure definition can contain structure variable declarations. For example, the employee's name and address could have been defined as individual structure types:

```
STRUCTURE /employee_name/
   CHARACTER*25  last_name
   CHARACTER*15  first_name
END STRUCTURE

STRUCTURE /employee_addr/
   CHARACTER*20  street_name
   INTEGER*2     street_number
   INTEGER*2     apt_number
   CHARACTER*20  city
   CHARACTER*2   state
   INTEGER*4     zip
END STRUCTURE
```

These become structure variables within the `employee_data` structure:

```
STRUCTURE /employee_data/
   RECORD  /employee_name/  name
   RECORD  /employee_addr/  addr
   INTEGER*4  telephone
   INTEGER*2  date_of_birth
   INTEGER*2  date_of_hire
   INTEGER*2  social_security(3)
   LOGICAL*2  married
   INTEGER*2  dependents
END STRUCTURE
```

Note that the reference to the first letter of the ninety-ninth employee's state of residence would now be `employees(99).addr.state(1:1)`.

A structure definition can also contain unions of maps. A map specifies that one or more variables are positioned contiguously in memory. The variables may be of any type, including structures, as shown below:

```
MAP
   INTEGER*4  manny, moe, jack
   CHARACTER*21  myname
END MAP
```

In the examples, the four-byte integers `manny`, `moe`, and `jack` appear first in memory, followed immediately by the 21-character variable `myname`. The starting addresses of each variable (and whether or not any bytes are used) is determined by the setting of the **$PACK** metacommand or /Zp command-line option.

Here is another example of a map:

```
MAP
   CHARACTER*7  yourname
   REAL*4  meg, joe, amy, zelda
END MAP
```

Maps can only appear within a **UNION** statement. When maps are combined in a union, the variables overlay each other as they do in an **EQUIVALENCE** statement, as shown in the following example:

```
UNION
   MAP
      INTEGER*4  manny, moe, jack
      CHARACTER*21  myname
   END MAP
   MAP
      CHARACTER*7  yourname
      REAL*4  meg, joe, amy, zelda
   END MAP
END UNION
```

Assuming that **$PACK:1** is in effect, the seven bytes of `yourname` will overlay the four bytes of `manny`, and the first three bytes of `moe`. The first byte of `meg` will overlay the last byte of `moe`, and so forth. If **$PACK:2** is in effect, `yourname` and `manny` plus `moe` will be overlaid as before, but the odd byte at the end of `yourname` will be left empty. Therefore, the first byte of `meg` will begin on the first byte of `jack`. For more information on the **$PACK** metacommand, see Chapter 6, "Metacommands."

Although individual structure elements may be written to or read from a file, any attempt to write a structure variable as a whole causes a compile-time error.

For more information, see the **RECORD** and **STRUCTURE** entries in Section 4.2, "Statement Directory."

1.5 Arrays

The number of elements in an array and the number of its dimensions are limited only by available memory. If the **$STRICT** metacommand is in effect, a warning is generated if more than seven dimensions are specified. An array element is referenced with the following 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 equal the number of dimensions in the array declaration. For information on declaring arrays, see the **DIMENSION** entry in Section 4.2, "Statement Directory."
	Each subscript must be an arithmetic expression. The result of the expression is converted to an integer by truncation. Each subscript must be an integer expression.(This limitation does not apply to Microsoft FORTRAN unless the **$STRICT** metacommand is in effect.) Function, array-element, and statement-element references are allowed. A subscript value can be positive, negative, or 0.

Examples

The following are 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)
```

Array elements are stored contiguously in memory so that array element *n+1* follows array element *n*, and so on. With multidimensional arrays, the elements are organized in a linear sequence because computer memory has only one dimension.

In FORTRAN, array elements are arranged in "column-major order." This means the order of storage is determined by incrementing the leftmost subscript first, then the next subscript to the right, and so on. The elements of the variable `array(3,4)`, are stored in the following sequence:

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

In Microsoft FORTRAN, array storage space can be allocated dynamically. That is, the size of each dimension can be set at run-time, rather than during compilation. This is done with an "allocatable" array.

For an array to be allocatable, it must have been declared with the attribute **ALLOCATABLE**, and the number of its dimensions specified. When the program runs, the **ALLOCATE** statement sets the size of each dimension. When the array is no longer needed, the **DEALLOCATE** statement returns the array's memory space to the common pool.

The following example shows the correct usage of the **ALLOCATABLE** attribute and the **ALLOCATE** and **DEALLOCATE** statements:

```
INTEGER  dataset[ALLOCATABLE](:,:,:)
INTEGER  reactor, level, points, error
DATA  reactor, level, points  / 10, 50, 100 /
.
.
.
ALLOCATE (dataset(reactor,level,points), STAT = error)
.
.
.
DEALLOCATE (dataset, STAT = error)
```

A significant enhancement to Microsoft FORTRAN 5.0 is the ability to perform operations on arrays as if they were ordinary scalar quantities. See Section 1.7.1, "Arithmetic Expressions," for more information.

1.6 Attributes

Attributes specify additional information about a variable, variable type, subprogram, or subprogram formal argument. Attributes allow your FORTRAN program, 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, in type declarations, and with the **INTERFACE** and **ENTRY** statements (see Section 4.2). Table 1.4 summarizes which attributes can be used with which objects. See Sections 1.6.1–1.6.12 for information on each attribute.

Table 1.4 Objects to Which Attributes Can Refer

Attribute	Variable and Array Declarations	Common-Block Names	Subprogram Specification and EXTERNAL Statements
ALIAS	Yes	Yes	Yes
ALLOCATABLE	Yes (arrays only)	No	No
C	Yes	Yes	Yes
EXTERN	Yes	No	No
FAR	Yes	Yes	Yes[1]
HUGE	Yes	No	No
LOADDS	No	No	Yes
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

object [[[*attrs*]]]

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

Examples

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

```
INTEGER intvar[REFERENCE, NEAR]
```

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

```
 INTERFACE TO INTEGER [C] FUNCTION PasFunc [PASCAL]
+(i, j, k)
 INTEGER [C] i, j, k
 END
```

1.6.1 ALLOCATABLE

This attribute specifies that an array is "allocatable"; that is, the size of each dimension is established dynamically at run-time, not during compilation.

The **ALLOCATABLE** attribute can appear either in a type declaration or in a **DIMENSION** statement. Each dimension of the allocatable array is indicated by a colon. If there is more than one dimension, the colons are separated by commas, as in this example which declares the allocatable array `dynamic` with three dimensions:

```
REAL*8  dynamic [ALLOCATABLE] (:,:,:)
```

The **ALLOCATABLE** attribute may not be applied to formal arguments, nor may it be used with the **NEAR** attribute. If you anticipate that an allocatable array's size will exceed 65,536 bytes, you must also declare the array with the **HUGE** attribute so the correct addressing will be generated.

For information on the use of allocatable arrays, see the reference sections on the **ALLOCATE** and **DEALLOCATE** statements.

1.6.2 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 1.3.6.1). 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.

Within the source file, the subprogram can be referred to only by the name given in its declaration. Outside the source file, the subprogram can be referred to only by its **ALIAS** name.

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

The **ALIAS** attribute overrides the **C** attribute. If the **C** attribute is used on a subprogram along with the **ALIAS** attribute, the subprogram will be given the

C calling convention, but not the C naming convention. It will instead receive the name given for the **ALIAS**, with no modifications.

The **ALIAS** attribute cannot be applied to formal arguments.

Example

This **SUBROUTINE** statement gives the subroutine `PasSub` the name `OtherName` outside this source file.

```
SUBROUTINE PasSub [ALIAS:'OtherName']
```

1.6.3 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 describes the differences between FORTRAN calling conventions and C calling conventions:

Difference	Explanation
Order in which parameters are pushed on the stack	Microsoft FORTRAN pushes 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).
Location of code that restores the stack when a procedure completes execution	In Microsoft FORTRAN this code is in the called procedure. This generates 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 formal argument is specified with the **REFERENCE** attribute. (Note that the **VARYING** attribute can only be specified 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 1.6.2).

When an integer variable is given the **C** attribute, it becomes a C integer variable. The default size for C and FORTRAN integers depends on the microprocessor. For example, on the 8086, Microsoft FORTRAN assumes 32-bit integers by default, while C assumes 16-bit. On other machines, both languages may assume 32-bit integers. The **C** attribute allows you to control integer size for compatibility between your FORTRAN programs and C functions.

Syntax

The C attribute cannot be applied to formal arguments, except those of **INTEGER** type, as in the following syntax line:

INTEGER[C] *argument*

The **ALIAS** attribute overrides the C attribute.

1.6.4 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 applied to formal arguments.

1.6.5 FAR

When used with formal arguments, the **FAR** attribute 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.

1.6.6 HUGE

The **HUGE** attribute is a convenient way to specify that a formal argument or an allocatable array may span more than one segment. (The **$LARGE** metacommand specifies the same thing.) The following two fragments behave identically:

```
FUNCTION Func (a[HUGE])
DIMENSION a(200)

$LARGE: a
FUNCTION Func (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 3.0 and earlier of Microsoft C do not support parameters with **HUGE** attributes.

1.6.7 LOADDS

The **LOADDS** attribute is applied only to separately compiled subprograms and functions. It directs the compiler to create a separate data segment for the data within that procedure; the base address (DGROUP) of this new data segment is automatically loaded into DS when the procedure is called. The use of a separate data segment permits the procedure's data to be called with 16-bit **NEAR** references rather than 32-bit **FAR** references. This speeds up data access. The default

data segment for the program is automatically reloaded when execution of the procedure terminates.

The **LOADDS** attribute is applied primarily to user-written routines that are to be included in an OS/2 dynamic link library (DLL). It is not needed for procedures that run in DOS programs because the FL command-line option /ND (name data segment) automatically assures that the new data segment's base address is loaded. The following is an example of the **LOADDS** attribute:

```
REAL*8 FUNCTION [LOADDS] GetNewData
```

1.6.8 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. This attribute can also be used with common blocks. Common blocks having the **NEAR** attribute are mapped into the default data segment.

Syntax

COMMON [[/[[*name*]] **[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 using the **NEAR** attribute
- Arrays specified using the **HUGE** attritute

- Arrays defined while the **$LARGE** metacommand is in effect
- Variables named in a **$LARGE** metacommand

1.6.9 PASCAL

The **PASCAL** attribute can be used with subprograms, common blocks, and formal argument type declarations (but not on formal arguments in the formal argument list). This attribute identifies an argument or subprogram as having the following 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

1.6.10 REFERENCE

The **REFERENCE** attribute can only be applied to formal arguments. It specifies that the argument's memory location is to be passed, rather than the argument's value. This is called "passing by reference."

1.6.11 VALUE

The **VALUE** attribute can only be applied to formal arguments. It specifies that the argument's value is to be passed, rather than the argument's memory location. This is called "passing by value."

If either the **C** or **PASCAL** attribute is specified on the subprogram definition, all arguments are assumed to have the **VALUE** attribute, since C and Pascal normally pass by value. Character values, substrings, assumed-size arrays, and adjustable-size arrays cannot be passed by value.

When a formal argument has the **VALUE** attribute, the actual argument passed to it can be of a different type. If type conversion is necessary, it is performed before the call, following the rules discussed in Section 1.7.1.2, "Type Conversion of Arithmetic Operands."

When passing by value, values cannot be returned through the formal argument, since the formal argument now addresses a stack location containing the value of the actual argument, rather than addressing the actual argument itself.

In C, arrays never pass 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 address returned by the **LOC**, **LOCNEAR**, or **LOCFAR** functions by value.

Example

Integer x is passed by value in the following example:

```
SUBROUTINE Subr (x[VALUE])
INTEGER x[VALUE]
```

1.6.12 VARYING

In FORTRAN, a formal argument must be defined for each actual argument. Other languages (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 conversion. When the **C** attribute is specified you can also apply the **VARYING** attribute, which permits the number of actual arguments to be different from the number of formal arguments. Actual arguments for which a formal argument is defined must still follow the type rules, however.

When writing a FORTRAN procedure with the **VARYING** attribute, be sure the procedure only uses arguments you actually passed, or you will get undefined results.

Note that the FORTRAN 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.

1.7 Expressions

An expression is a formula for computing a value. Expressions consist of operands and operators. The operands can be function references, variables, structure elements, 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, as shown below:

Expression	Result
Arithmetic	Integer, real, or complex value
Character	Character or string value
Relational	Logical value
Logical	Logical or integer value

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

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

Any variable, array, array element, structure, structure element, or function that is referred to in an expression must be defined at the time the reference is made, or the results are undefined. Integer variables must have a numerical 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 an expression multiplies `(37.8 / scale**expo + factor)` by zero, then `(37.8 / scale**expo + factor)` may not be evaluated. Similarly, if a false value and a logical expression are operands of the **.AND.** operator, the expression may not be evaluated. In the following example, the expression `(SWITCH .EQ. on)` may not be evaluated:

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

Microsoft FORTRAN permits expressions to use full arrays in the same way they would normally use single (scalar) arguments. For a complete explanation, see Section 1.7.5, "Array Expressions."

1.7.1 Arithmetic Expressions

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

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

- Array references
- Structure-element references

Other arithmetic expressions are built from the basic operands in the preceding list, using parentheses and the arithmetic operators shown in Table 1.5.

Table 1.5 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 are binary operators, appearing between two operands. The plus and minus operators can also be used as unary operators, which precede a single operand.

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 does not allow two arithmetic operators to appear consecutively. For example, FORTRAN prohibits `first**-second`, but permits `first**(-second)`.

The following list shows examples of the precedence of arithmetic operators:

Expression	Equivalent Expression
`3 * 7 + 5`	`(3 * 7) + 5`
`-one**two`	`-(one**two)`
`+x / y`	`+(x / y)`
`area / g - qu**2**fact`	`(area / g) - (qu**(2**fact))`

The following arithmetic operations are prohibited:

- Division by zero
- Raising a zero-value operand to a negative power
- Raising a negative-value operand to a nonintegral real power

1.7.1.1 Integer Division

When one integer is divided by another, the truncated quotient of the two operands is returned. Thus, 7/3 evaluates to 2, (–7)/3 evaluates to –2, and both 9/10 and 9/(–10) evaluate to zero.

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 it. The result is 0. The assignment, therefore, sets `X` equal to zero.

1.7.1.2 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 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. **DOUBLE COMPLEX** or **COMPLEX*16** (highest rank)
2. **COMPLEX[[*8]]**
3. **DOUBLE PRECISION** or **REAL*8**
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** operand is first converted to **REAL*4**. The result of the operation is also a value of data type **REAL*4**. Similarly, in 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 examples show 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 1.7.1.1) is performed on `i1` and `i2`. The quotient 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 added to `c2`.

For expressions with integer operands, the type of the result is controlled by the types of the operands and the setting of the **$STORAGE** metacommand, the result is of whichever type is larger. For example, if you declare `j` and `k` as **INTEGER*2** variables, and use the **$STORAGE:4** metacommand, the result of the expression `j + k` is of type **INTEGER*4**. (The same result occurs if the **$STORAGE** metacommand is omitted, since the default for **$STORAGE:***n* is 4.)

Note that the compiler usually removes higher-precision arithmetic when optimizing if it will not affect the result and if **$DEBUG** is not set. For example, the intermediate value obtained when adding `j` to `k` is probably 16-bit, even if **$STORAGE:4** were specified as in the following example:

```
INTEGER*2  i, j, k
i = j + k
```

Using **$STORAGE:4** does not affect **INTEGER*2** expressions which have only the plus (+), minus (–), or multiplication (*) operators in them. The results are the same because conversion to an **INTEGER*4** intermediate result does not prevent overflows in **INTEGER*2** arithmetic. (Note that such overflows are not reported unless the **$DEBUG** metacommand is in effect.)

Table 1.6 shows how arithmetic operands are converted from one data type to another.

Table 1.6 Arithmetic Type Conversion

Data Type	Converting to the Next-Highest-Ranked Data Type	Converting to the Next-Lowest-Ranked Data Type
DOUBLE COMPLEX or **COMPLEX*16**	(Highest rank.)	Convert imaginary and real parts, individually, from **REAL*8** to **REAL*4**.[a]
COMPLEX*8	Convert imaginary and real parts, individually, from **REAL*4** to **REAL*8**.	Delete imaginary part.
DOUBLE PRECISION or **REAL*8**	Convert to **REAL*4**, and add a 0.0 imaginary part.[a]	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	Use as least-significant part, and set sign bit in most-significant part.	(Lowest rank.)

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

1.7.2 Character Expressions

A character expression produces a value that is of type character. There are six 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
6. Character structure-element references

The concatenation operator (//) is the only character operator. It is used as follows:

first//second

This produces a character string consisting of 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 `'AB CDE'`:

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

When two or more string variables are concatenated, the resulting string is as long as the declared lengths of the string variables. Leading and trailing blanks are not discarded. For example:

```
CHARACTER*10  first
CHARACTER*6   second
first = 'heaven'
second = ' sent'
WRITE (*, *) first//second
```

The result is the 16-character string `'heaven      sent '`. Note that there is a total of five spaces between `'heaven'` and `'sent'`.

If you concatenate C strings, remember a null character (**\0**) is automatically appended to each C string. For example, the expression

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

is equivalent to the following C string:

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

1.7.3 Relational Expressions

Relational expressions compare the values of two arithmetic or character expressions. You cannot compare an arithmetic variable with a character variable.

In Microsoft FORTRAN, an arithmetic expression can be compared with a character expression. The arithmetic expression is treated as if it were a character expression (that is, a sequence of byte values). The two expressions must be identical on a byte-by-byte basis, or they are not equal.

For example, if `'A'` were assigned to a four-byte integer, the ASCII value of the letter A (hex 41) would be the variable's least-significant byte, and the other bytes would be zeros. If `'A'` were assigned to a character variable four characters long, the ASCII value of the letter A (hex 41) would be the variable's most-significant byte because character variables are left-justified. Therefore, the two variables would not be equal, even though they held the same nominal value.

A relational expression produces a result of type **LOGICAL** (**.TRUE.** or **.FALSE.**). Relational expressions can use any of the operators shown in Table 1.7 to compare values.

Table 1.7 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 relational operators are binary operators and appear between their operands. A relational expression cannot contain another relational expression, so there is no relative precedence or associativity among the relational operands. The following program fragment is therefore invalid:

```
REAL*4  a, b, c, d
IF ((a .LT. b) .NE. c)  d = 12.0
```

Assume that `a` is less than `b`. After the first part of the expression is evaluated, the expression is

```
.TRUE. .NE. c
```

However, `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 (as discussed in Section 1.7.4).

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. One operand is less than another if it appears earlier in the collating sequence. For example, the expression `('apple'.LT.'banana')` returns the value **.TRUE.**, and the expression `('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.

1.7.4 Logical Expressions

A logical expression produces a logical value. There are seven 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
6. Integer constants or variables
7. Logical structure-element references

Other logical expressions are constructed from the basic operands in the preceding list by using parentheses and the logical operators of Table 1.8.

Table 1.8 Logical Operators

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

The **.AND.**, **.OR.**, **.XOR.**, **.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` is **.TRUE.**, then `(.NOT. switch)` is **.FALSE.**.

Logical operators allow only arguments of the **LOGICAL** type. Microsoft FORTRAN also permits integer arguments, which may be integer constants, integer variables, integer structure elements, or integer expressions. Operations are "bitwise." For example, the expression `k .XOR. m` performs an "exclusive-or" comparison on matching bits in the operands, and sets or clears the corresponding bit in the integer value it returns. If both operands are not of the same integer type, the lower-precision operand is converted to the higher-precision type.

Note that the result of comparing two integer expressions with a logical operator is of **INTEGER** type, not **LOGICAL**.

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

Two **.NOT.** operators cannot be adjacent, 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 **.TRUE.**. Table 1.9 shows the values of logical expressions.

Table 1.9 Values of Logical Expressions

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

Examples

The following example 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)
```

The following example demonstrates the use of integers in logical expressions to perform byte masking:

```
INTEGER*2  lowerbyte, dataval, mask
mask = #00FF        ! mask the most-significant byte
dataval = #1234
lowerbyte = (dataval .AND. mask)
WRITE (*, '('' '', 2Z4)') dataval, lowerbyte
```

The output is as follows:

```
1234  34
```

1.7.5 Array Expressions

Microsoft FORTRAN permits operations on full arrays that would normally only work with single arguments (scalars). For example, two arrays may be added element-by-element using only the addition operator (+). Every element in an array can be divided by a constant value. Two arrays of **LOGICAL** or **INTEGER** variables can be compared with logical operators such as **.AND.** or **.GE.**, and so on. Arrays can also be passed to functions (both intrinsic and external), with the function operating on each element and returning an array of the results.

When two or more array operands appear in an assignment statement or expression, they all must "conform." That is, they must have the same number of dimensions, and corresponding dimensions must be the same size and have the same upper and lower boundaries. For example, the arrays `first(6,3)` and `second(3,2,3)` do not conform. Although they take up the same amount of memory, their dimensions do not match. The arrays `third(3,4)` and `fourth(-1:1,5:8)` do not conform either. They have the same number of dimensions, and each is of the same size, but the dimension boundaries do not match. Any attempt to combine non-conforming arrays in an arithmetic expression, or assign one to another, results in a compile-time error. Note that a scalar quantity conforms to any array.

If arrays are to conform, the sizes of their dimensions must be fully specified at compile time. Therefore, adjustable and allocatable arrays do not conform; they cannot be combined in expressions either with themselves or fixed arrays, even if their dimensions match.

All operations that are permitted for scalar arguments are permitted for array arguments. An array expression may be the right-hand term of an assignment statement whose left-hand term is an array. Array expressions may only appear in assignment statements and as function arguments. For example, if `array1`, `array2`, `array3`, and `logarray` are conforming arrays, then all the following statements are legal:

```
array1 = 42                     !Assign constant to each element.
array3 = array 1 / array2       !Divide corresponding element.
array2 = - array1               !Negate each array1 element.
array1 = array2 - 3             !Subtract constant from each element.
array1 = array2 - array3(7)     !Subtract value of array3(7) from each element.
array2 = array3 ** array1       !Raise each element to power of
                                !corresponding element.
array3 = MyFunc (array2)        !Apply MyFunc to each element.
logarray = array1 .LT. array2   !Logically compare corresponding elements.
logarray = .NOT. array2         !Invert bits of each array2 element.
```

All intrinsic functions that take scalar arguments may also take array arguments. If you wish to pass array arguments to external functions that take scalar arguments, you must declare the external function in an **INTERFACE TO** statement. None of the function's formal arguments may be arrays.

NOTE *In processing array expressions, the compiler may generate a less efficient sequence of machine instructions than it would if the arrays were processed in a conventional* ***DO*** *loop. If execution speed is critical, it may be more efficient to handle arrays element-by-element.*

1.7.6 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 2 Program Structure

This chapter explains the structure of FORTRAN programs: what can be included and in what sequence. Information on line types and statement labels is provided, and restrictions on the order of statements and metacommands are discussed. This chapter also includes information on arguments and on the program units processed by the FORTRAN compiler: main programs, subroutines, functions, and block-data subprograms.

2.1 Lines

The position of characters within FORTRAN lines is significant. The following list shows how column position determines character interpretation:

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	Description

Metacommand lines	A metacommand line has a dollar sign ($) in column 1.
	
	Metacommands control operation of the Microsoft FORTRAN Compiler. (See Chapter 6 for more information.) The following lines are metacommands:
	
	`$DEBUG:'pdq'`
	`$DECLARE`
	`$LINESIZE:132`

Initial lines	The first (or only) line of a FORTRAN statement is called the "initial line." An initial line has either a blank or a zero in column 6, and has either blanks or a statement label in columns 1 through 5. (The only exception to this rule is the statement which follows a logical **IF** statement.) The following are initial lines:
	
	`      GOTO 100`
	`0002  CHARACTER*10 name`
	`100   0CONTINUE`
	`  1000STOP ' '`

Continuation lines	A continuation line has blanks in columns 1 through 5 and a character (other than a blank or a zero) in column 6. A continuation line increases the room available to write a statement. A statement may be extended to include as many continuation lines as memory allows. The compiler issues a warning if a statement has more than 19 continuation lines. The second line below is a continuation line (this limitation does not apply to Microsoft FORTRAN unless the **$STRICT** metacommand is in effect):
	
	`C     Sample Continuation line`
	`      INTEGER*4  count, popu, local,`
	`     +          ovrflo, incrs, provnc`

Comment lines	A comment line generally has an uppercase or lowercase C or an asterisk (*) in column 1 or is entirely blank. An exclamation point appearing as the first nonblank character also indicates a comment line in Microsoft FORTRAN. (A line with an exclamation point in column 6 is interpreted as a continuation line.)

Comment lines do not affect the execution of the FORTRAN program in any way. Comment lines can appear within statements that have continuation lines. Debug lines (described below) are sometimes treated as comment lines. For more information, see the **$DEBUG** and **$NODEBUG** entry in Section 6.2, "Metacommand Directory." The following are examples of comment lines:

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

Comments can also be added to program lines, following the statement. In this case, the comment must begin with an exclamation point (**!**).

```
hyp = SQR (a**2 + b**2)   !  hypotenuse
```

Debug lines

In Microsoft FORTRAN, any line with an uppercase or lowercase letter in the first column is considered a comment line. When a letter (except C or c) is specified in a **$DEBUG** metacommand, that letter is removed from all succeeding lines that include it in column 1, and the remainder of the line is compiled. The case of the letter (either in the **$DEBUG** metacommand or the program) is not significant. See the **$DEBUG** and **$NODEBUG** entry in Section 6.2 for more information on debug lines. The following lines are examples of debug lines:

```
B     RETURN 1
z     WRITE(*, *) count
```

2.2 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 are ignored. (For example, the labels `23 4` and `234` are identical.) A label may not be repeated.

2.3 Free-Form Source Code

The **$FREEFORM** metacommand lets you enter source code without most of the restrictions imposed by the conventional FORTRAN format. Most of the rules in Section 2.1, "Lines," do not apply to the free-form format. See the **$FREEFORM** and **$NOFREEFORM** entry in Section 6.2, "Metacommand Directory," for more information on free-form format.

The following rules define the free-form format:

- A double quotation mark (") in column 1, or an exclamation point as the first nonblank character, indicates a comment line.
- Initial lines may start in any column.
- An initial line may begin with a statement label of from one to five digits (blanks are ignored). Statement labels need not be followed by blanks to separate them from the statement; the first nonblank, nonnumeric character is the beginning of the statement.
- If the last nonblank character of a line is a minus sign, it is discarded and the next line is considered a continuation line. The continuation line may start in any column.
- Characters in columns 81 and higher are ignored.
- Alphabetic characters and asterisks are not allowed as comment markers in column 1.

2.4 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 4, "Statements," defines each Microsoft FORTRAN statement.

The ANSI standard for FORTRAN prescribes the ordering of the statements and lines in a FORTRAN program unit. Some of Microsoft FORTRAN's extensions have additional requirements. Figure 2.1 shows which statements and metacommands must precede and which must follow any specific statement or metacommand.

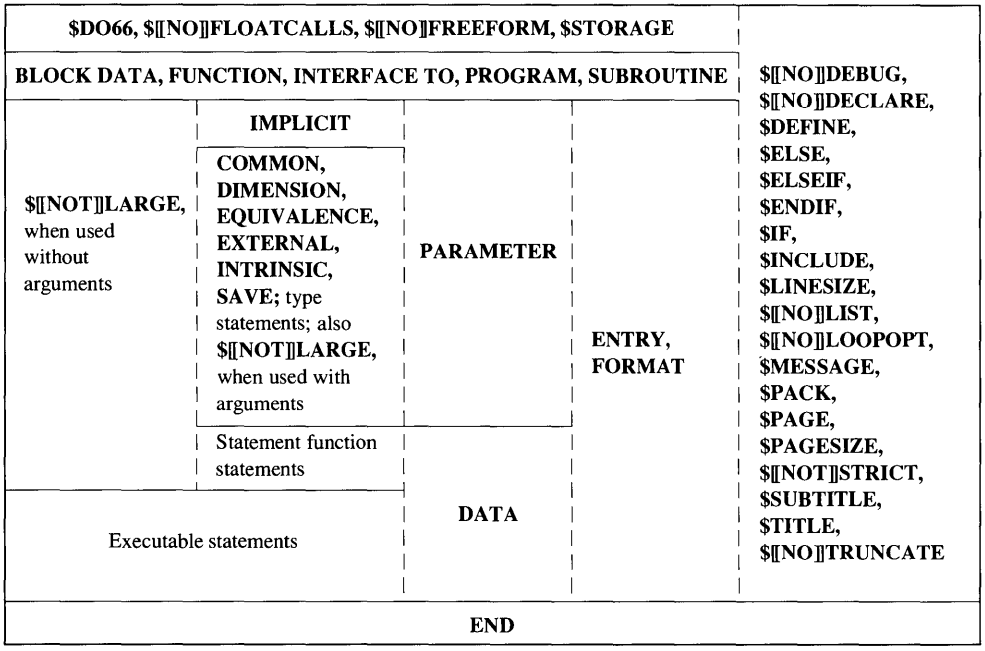


Figure 2.1 Order of Statements and Metacommands

Suppose your program contains program elements *a* and *b*. In Figure 2.1, if the box containing element *a* is above the box containing element *b*, then *a* must appear before *b* in your program. An **IMPLICIT** statement, for example, must appear before a **COMMON**, **DATA**, or **END** statement, and so on.

If, in Figure 2.1, *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 boxes with **DATA**, **PARAMETER**, **IMPLICIT**, **COMMON**, and statement-function statements, and so on. 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 2.1:

- 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.
- The **BLOCK DATA**, **FUNCTION**, **INTERFACE TO**, **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 4.1, "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 TO** statements must precede references to the subprograms they define.
- The **$DO66**, **$[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 declarations section. Other metacommands can appear anywhere.
- Block-data subprograms may not contain statement function statements, **FORMAT** statements, or executable statements.

2.5 Arguments

"Arguments" are the 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. The "actual argument" is the specific variable, expression, array, function name, or other item passed to a subroutine or function when it is called.

The number of actual arguments must be the same as the number of formal arguments (unless the **VARYING** attribute is specified), and the corresponding types must agree. If a procedure is called more than once in a program unit, the compiler checks that the number and types of actual arguments are the same in each call (consistency). If a procedure was defined prior to its first use, or in an **INTERFACE TO** statement, the compiler also checks that the number and types of the actual arguments match the number and types of the formal arguments (validity).

Arguments normally pass values into and out of subroutines or functions by reference (they pass the memory address of the argument). In Microsoft

FORTRAN, this is the default. You can use the **VALUE** attribute (described in Section 1.6.11) to pass arguments by value.

Upon entry to a subroutine or function, the actual arguments are associated with the formal arguments. This association remains in effect until the subroutine or function terminates execution. 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 alters 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
      PARAMETER  (header = 'Specific Gravity')
      DATA  grav / 2.8327 /
      WRITE (*, *) header, grav
C     Header is an actual argument:
      CALL Report (header, grav)
      WRITE (*, *) header, grav
      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; it assigns a value
C     to a formal argument that corresponds to a constant:
      title = 'Density (kg/cubic m)'
      WRITE (*, *) title, data
      END
```

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

If an actual argument is an expression, it is evaluated before the association of formal and actual arguments. If an actual argument is an array element, its subscript expressions are evaluated before the association. The subscript expressions remain constant throughout the execution of the subroutine or function, even if they contain variables that receive new values during the execution of the subroutine or function.

The following list shows how actual and formal arguments can be associated:

Actual Argument	Formal Argument
A variable, an array element, a structure element, or an expression.	Variable 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. A reference to an out-of-bounds element is not detected as an error, and has unpredictable results.	Array name.
An array. Each element of the array is passed to the procedure, one element at a time, and the procedure is executed once for each element.	A variable. Its procedure must be declared in an **INTERFACE TO** statement, or it must be an intrinsic function, if arrays are to be passed to a scalar formal argument.
An alternate-return specifier (**label*) in the **CALL** statement. The same label can be used in more than one alternate-return specifier.	An asterisk (*).
The name of an external subroutine or function, or intrinsic function. The 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 first be declared with an **INTRINSIC** statement in the program unit where it is used as an actual argument.	Any unique name which is used in a subroutine call or function reference within the procedure.

The following intrinsic functions may not be associated with formal subroutine arguments or function arguments:

ALLOCATED	**EPSILON**	**LEN_TRIM**	**MIN**
AMAX0	**FLOAT**	**LGE**	**MIN0**
AMAX1	**HFIX**	**LGT**	**MIN1**
AMIN0	**HUGE**	**LLE**	**MINEXPONENT**
AMIN1	**ICHAR**	**LLT**	**NEAREST**
CHAR	**IDINT**	**LOC**	**PRECISION**
CMPLX	**IFIX**	**LOCFAR**	**REAL**
DBLE	**IMAG**	**LOCNEAR**	**SCAN**
DCMPLX	**INT1**	**LOG**	**SNGL**
DFLOAT	**INT2**	**LOG10**	**TINY**
DMAX1	**INT4**	**MAX**	**VERIFY**
DMIN1	**INTC**	**MAX0**	
DREAL	**INT**	**MAX1**	
EOF	**JFIX**	**MAXEXPONENT**	

When passing integer arguments by reference, an **INTEGER*2** variable cannot be passed to an **INTEGER*4** formal argument, and an **INTEGER*4** variable cannot be passed to an **INTEGER*2** formal argument. You must convert the data type using the intrinsic functions **INT4** or **INT2** (described in Chapter 5, "Intrinsic Functions and Additional Procedures"). Note that the result of the conversion is stored at a temporary memory location, so the subroutine can no longer assign a new value to the actual 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:

```
C     This is incorrect and produces an error:
$STORAGE:4
      INTEGER*2 j, k
      CALL Subr(j + k)

      SUBROUTINE Subr (n)
      INTEGER*2 n
      .
      .
      .
      END
```

An error occurs because the result of the expression `j+k` is of the type **INTEGER*4**. You must write the subroutine call as

```
      CALL Subr (INT2(j + k))
```

Integer arguments passed by value are not subject to the same restrictions. The conversion rules for value arguments are the same as the conversion rules described in Table 1.6, "Arithmetic Type Conversion." For example, you can pass a real value to an integer argument because the conversion is performed automatically.

2.6 Program Units

The 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 2.7 through 2.10):

Program Unit	Description
Main program	Any program unit that does not have a **SUBROUTINE**, **FUNCTION**, 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 is called from other program units with 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.

The **PROGRAM**, **SUBROUTINE**, **BLOCK DATA**, **FUNCTION**, and Statement Function statements are described in detail in Section 4.2, "Statement Directory." Related information is provided in the entries for the **CALL** and **RETURN** statements.

Subprograms make it easier to develop large, well-structured programs, especially in the following situations:

Situation	Result
You have a large program	You can more easily develop, test, maintain, and compile a large program when it is divided into parts.
You intend to include certain subprograms in more than one program	You can create object files that contain these subprograms and link them to the programs in which they are used.
You anticipate altering a procedure's implementation	You can place the procedure in its own file and compile it separately. You can change the procedure or even rewrite it in assembly language, Microsoft Pascal, or Microsoft C. The rest of your program does not need to change.

2.7 Main Programs

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, the use of "main" is likely to cause confusion and 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**.

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

For further information about programs, see the **PROGRAM** entry in Section 4.2, "Statement Directory."

2.8 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. The subroutine then returns control to the statement immediately following the one that called it, or to a statement specified as an alternate return. See the **CALL** entry in Section 4.2, "Statement Directory," 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 the **SUBROUTINE** entry in Section 4.2, "Statement Directory," for further information.

2.9 Block-Data Subprograms

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

Variables are normally initialized with **DATA** statements. Variables in named-common blocks can be initialized only in block-data subprograms. Variables in the blank-common block cannot be initialized in block-data subprograms. See the **BLOCK DATA** entry in Section 4.2, "Statement Directory," for more information.

2.10 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

External and statement functions are described in more detail below. Intrinsic functions are described in Chapter 5, "Intrinsic Functions and Additional Procedures".

An arithmetic, character, or logical expression may reference a function. Executing the function reference evaluates the function, and the returned value is used as an operand in the expression with the function reference. Note that the length of a character function must be specified by an integer-constant expression.

The syntax of a function reference is as follows:

fname ([[*argument-list*]])

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

Function arguments follow the same rules as those for subroutines (except that alternate returns are not allowed); these are described in the **CALL** entry in Section 4.2, "Statement Directory." Some additional restrictions specific to statement functions and intrinsic functions are described in Section 2.10.2, "Statement Functions," and Chapter 5, "Intrinsic Functions and Additional Procedures," respectively.

2.10.1 External Functions

External functions are user-defined functions (as opposed to FORTRAN's intrinsic functions). They may be included with the main program source code or in a separately compiled unit. They begin with a **FUNCTION** statement and conclude with an **END** statement.

Example

```
C      The following external function calculates
C      the Simpson approximation of a definite 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
          ELSE IF (MOD (i, 2) .EQ. 0) THEN
             factor = 2
          ELSE
             factor = 4
          END IF
          sum = factor * y(i) + sum
100    CONTINUE
       Simpson = INT ((REAL (delx) /3.0) * REAL (sum))
       END
```

2.10.2 Statement Functions

A statement function is defined by a single statement and is similar in form to an assignment statement. The body of a statement function defines the statement function.

A statement function must be the first nondeclaration statement in a program unit, but it is not executed at that point; it is a nonexecutable statement. It is executed, like other functions, by referencing it in an expression.

For information on the syntax and use of a statement function statement, see the Statement Function entry in Section 4.2, "Statement Directory."

CHAPTER 3

The Input/Output System

This chapter explains FORTRAN's input/output (I/O) system. The chapter introduces FORTRAN's file types and summarizes the I/O statements available to manipulate data within them. Additional sections describe options available in FORTRAN I/O statements and the use of carriage-control characters, formatted I/O, and edit descriptors. List-directed and namelist-directed I/O are also discussed.

3.1 The FORTRAN I/O System

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

Type	Description
External files	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 variable, character array, character array element, character structure element, character substring, or noncharacter array. For information on internal files, see Section 3.5.

All files consist of records, which are sequences of characters or values. See *Microsoft FORTRAN Advanced Topics* for information on the format of records.

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.

3.2 I/O Statements

I/O statements transfer data, manipulate files, or determine the properties of the connections to files. Table 3.1 lists the I/O statements.

Table 3.1 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 the beginning
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 5.1.14, "End-of-File Function."* ***EOF*** *returns a logical value that indicates whether there is 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 three options of the **CLOSE** statement are [[, **ERR=***errlabel*]], [[, **IOSTAT=** *iocheck*]], and [[, **STATUS=***status*]]. (Although the **UNIT=** descriptor is optional, the *unitspec* itself is not.)

The **ERR=** and **IOSTAT=** options control error handling, and the **STATUS=** option specifies whether to keep or delete the file after disconnecting. The following statement closes the file connected to unit 2, transfers execution to the statement labeled `100` if there is an I/O error, and uses the default value of the **STATUS=** option:

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

Options used in only one statement are described in Section 4.2, "Statement Directory." Fourteen of the options in I/O statements, however, are used in more than one I/O statement. Table 3.2 lists these options, the I/O statements they are used in, and the section that describes each.

Table 3.2 I/O Options

Option	I/O Statements	Section
ACCESS=*access*	**INQUIRE, OPEN**	3.2.3, "File Access Method"
BLOCKSIZE=*blocksize*	**INQUIRE, OPEN**	3.2.4, "Input/Output Buffer Size"
editlist	**FORMAT, PRINT, READ, WRITE**	3.2.5, "The Edit List"
ERR=*errlabel*	All except **PRINT**	3.2.6, "Error and End-of-File Handling"
FILE=*file*	**INQUIRE, OPEN**	3.2.1, "File Names"
[**FMT=**] *formatspec*	**PRINT, READ, WRITE**	3.2.7, "Format Specifier"
FORM=*form*	**INQUIRE, OPEN**	3.2.9, "File Structure"
iolist	**PRINT, READ, WRITE**	3.2.10, "Input/Output List"
IOSTAT=*iocheck*	All except **PRINT**	3.2.6, "Error and End-of-File Handling"
MODE=*mode*	**INQUIRE, OPEN**	3.2.11, "File Sharing"
[**NML=**]*nmlspec*	**PRINT, READ, WRITE**	3.2.8, "Namelist Specifier"
REC=*rec*	**LOCKING, READ, WRITE**	3.2.12, "Record Number"
SHARE=*share*	**INQUIRE, OPEN**	3.2.11, "File Sharing"
[**UNIT=**]*unitspec*	All except **PRINT**	3.2.2, "Units"

The following list briefly summarizes the options listed in Table 3.2. Sections 3.2.1–3.2.12 provide a complete discussion of these options. The **FILE=**, **UNIT=**, and **ACCESS=** options are presented in the order you must use them, followed by the other options listed alphabetically.

Option	Description
ACCESS	Specifies the method of file access, which can be **'SEQUENTIAL'**, **'DIRECT'**, or **'APPEND'**. Returns the method of file access, which can be **'SEQUENTIAL'** or **'DIRECT'**.
BLOCKSIZE	Specifies or returns the internal buffer size used in I/O.

editlist	Lists edit descriptors. Used in **FORMAT** statements and format specifiers (the **FMT=***formatspec* option) to describe the format of data.
END	Transfers control to the statement with the label in the **END=** option when end-of-file is encountered.
ERR	Controls I/O error handling. Specifies the label of an executable statement where execution is transferred after an error.
IOSTAT	Controls I/O error handling. Specifies a variable whose value indicates 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 either **'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'.**
NML	Specifies a *namelist* group to be input or output.
REC	Specifies the first (or only) record of a file to be locked, read from, or written to.
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'.**
UNIT	Specifies the unit to which a file is connected.

3.2.1 File Names (FILE=)

File names must follow the rules listed in Section 1.2, "Names." The name of an internal file is the name of the character variable, character array, character array element, character structure element, character substring, or noncharacter array that makes up the file. The name of an external file must be a character string that the operating system recognizes as a file name. The operating system assumes the file is in the current working directory if you do not specify a path. External file names must follow the file-naming conventions of your operating system, as well as the rules listed in Section 1.2, "Names." Wild-card specifications are not permitted.

An external file can be connected to a FORTRAN unit number 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 for a unit other than one of the preconnected units (0, 5, or 6) and no name is specified in the **OPEN** statement, the file is considered a scratch or temporary file, and a default name is used.
- 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 entered by the user, as described in the **OPEN** entry in Section 4.2, "Statement Directory."
- 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 the **READ** and **WRITE** entries in Section 4.2, "Statement Directory." 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 the **OPEN** entry in Section 4.2, "Statement Directory ."

3.2.2 Units (UNIT=)

For most I/O operations, a file must be identified by a unit specifier. The unit specifier for an internal file is the name of that internal file (see Section 3.5 for information on internal files). For an external file, a file is connected 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** (which always writes to standard output), a **READ** statement that contains only an I/O list and format specifier (which always reads from standard input), and an **INQUIRE** by file (which specifies the file name, rather than the unit to which the file is connected).

An external unit specifier must be either an integer expression or an asterisk (*). The integer expression must be in the range –32,767 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 causes a compile-time error. Units 0, 5, and 6, however, can be connected to any file with 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 the **READ** and **WRITE** entries in Section 4.2, "Statement Directory."

Examples

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

```
C     The output of this program is FM004.DAT
      CHARACTER*14 fname
      ifile = 4
      WRITE (fname, 100) ifile
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
C     to an external file named 'COSINES':
      OPEN (UNIT = 2*3, FILE = 'COSINES', STATUS = 'NEW')
      DO 200 a = 0.1, 6.3, 0.1
         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'')')
      END
```

3.2.3 File Access Method (ACCESS=)

The following sections describe the three methods of file access: sequential, direct, and appended. Sequential files contain data recorded in the order in which it was written to the file; direct files are random-access files.

3.2.3.1 Sequential File Access

The records in a sequential file appear in the order in which they were written to the file.

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.

It is not possible to access a particular sequential record directly because all the preceding records must be read through first. Direct operations to files opened for sequential access are not allowed. An existing sequential file can be opened in **'APPEND'** mode. The file is positioned immediately after the last record and before the end-of-file mark.

3.2.3.2 Direct File Access

Direct files are random-access files whose records can be read or written to 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. 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.

Except for binary files, the number of bytes written to a record must be less than or equal to the record length. For binary files, a single **READ** or **WRITE** statement reads or writes as many records as needed 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.

Positioning a direct-access file past the end-of-file marker and attempting to read from it can cause a run-time error. Reading an unwritten record from a direct-access file is also illegal and can cause 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.

Each **READ** or **WRITE** operation on a direct-access file must explicitly specify the record to be accessed. Microsoft FORTRAN also permits sequential operations on files opened for direct access; the operation takes place on the next record.

The following statements 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
```

A file created as a direct-access file and opened in **'APPEND'** mode defaults to sequential access.

3.2.4 Input/Output Buffer Size (BLOCKSIZE=)

Many programs are "I/O bound," meaning their speed of execution is largely determined by the speed of I/O operations. I/O speed, in turn, is influenced by the size of the I/O buffer. In general, the larger the buffer, the faster the I/O, since a large buffer reduces the total number of reads and writes needed to transfer a given amount of data.

The value of **BLOCKSIZE** is an integer expression specifying the internal buffer size for use in I/O. See *Microsoft FORTRAN Advanced Topics* for specific information about block sizes and using the **BLOCKSIZE=** option.

3.2.5 The Edit List

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

The edit list (including the outer parentheses) is a character constant, and is enclosed in apostrophes when it appears in a **READ** or **WRITE** statement. In Microsoft FORTRAN, the edit list may be enclosed in quotes. There are no apostrophes around the edit list when it appears in a **FORMAT** statement.

The edit list is a series of formatting descriptors separated by commas. You may omit the comma between two 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 any resulting ambiguity.

There are three types of editing descriptors:

1. Nonrepeatable edit descriptors (*nonrepeatable*)
2. Repeatable edit descriptors optionally preceded by a repeat specification (*n-repeatable*)
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 3.7.1, "Nonrepeatable Edit Descriptors," and 3.7.2, "Repeatable Edit Descriptors." The following list gives examples of each type of editing descriptor:

Editing Descriptor	Examples

nonrepeatable	`'Total = '`
	`3Hyes`
	`SP`
	`BN`
	`1X`

n-repeatable	`I5`
	`2I5`
	`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 next FORMAT statement is below the
C     hyphens:   ----------------------
100   FORMAT (' A = ', I5, 'B = ', I5)
      WRITE (*, 200) x, y
C     The editlist in the next FORMAT statement is below the
C     hyphens:   -----------------
200   FORMAT (' ', 2(F8.3, 1Hm))
      END
```

The program above produces the following output:

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

Note that each formatted **WRITE** statement writes an initial blank to the terminal device as a carriage-control character.

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

If an error occurs or the end-of-file record is encountered during an I/O operation, the action taken depends on the presence and definition of the **ERR=** *errlabel*, **IOSTAT=***iocheck*, and **END=***endlabel* options. Note that the **READ** statement is the only I/O statement that does not consider reaching an end-of-file record as an error.

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

Table 3.3 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 items in *iolist* become undefined.

Table 3.3 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 occurs.	Run-time error occurs.
No	No	Yes	Program terminates.	Go to *errlabel*.
No	Yes	No	Go to *endlabel*.	Run-time error occurs.
No	Yes	Yes	Go to *endlabel*.	Go to *errlabel*.

Table 3.3 *(continued)*

IOSTAT Set	END Set	ERR Set	End-of-File Occurs	Error, or Error and End-of-File, Occurs
Yes	No	No	Set *iocheck* = –1 and continue.	Set *iocheck* = run-time error number and continue.
Yes	No	Yes	Set *iocheck* = –1 and continue.	Set *iocheck* = run-time error number and go to *errlabel.*
Yes	Yes	No	Set *iocheck* = –1 and go to *endlabel.*	Set *iocheck* = run-time error number and continue.
Yes	Yes	Yes	Set *iocheck* = –1 and go to *endlabel.*	Set *iocheck* = run-time error number 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**:

Situation	Result
Neither *errlabel* nor *iocheck* is present	The program is terminated, and a run-time error message is given.
Only *errlabel* is present	Control is transferred to the statement at *errlabel.*
Only *iocheck* is present	The value of *iocheck* is set to the run-time error number and control is returned as if the statement had executed without error.
Both *errlabel* and *iocheck* are present	The value of *iocheck* is set to the run-time error number and control is transferred to the statement at *errlabel.*

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

Examples

In the following program, none of the available options (**ERR=**, **IOSTAT=**, or **END=**) are set. Therefore, 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
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   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
      WRITE (*, *) 'Please enter i:'
50    READ  (*, *, ERR = 100, IOSTAT = j) i
      WRITE (*, *) 'This is i:', i, ' iostat = ', j
      STOP ' '
100   WRITE (*, *) 'iostat = ', j, ' Please enter new i:'
      GOTO 50
      END
```

3.2.7 Format Specifier (FMT=)

Format specifiers either contain an edit list or specify the statement label of the format to be used. Format specifiers are used in **PRINT**, **READ**, and **WRITE** statements.

The following sections show the seven acceptable types of format specifiers and provide an example of how each format specifier can be used.

3.2.7.1 FORMAT Statement Label

A format specifier can be the label of a **FORMAT** statement. The edit list in the **FORMAT** statement becomes the format for the data. The following syntax shows how a **FORMAT** statement label can specify an edit list for a **WRITE** statement:

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

3.2.7.2 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 the **ASSIGN** entry in Section 4.2, "Statement Directory."

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

3.2.7.4 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
```

3.2.7.5 Asterisk (*)

When an asterisk (*) is used as a format specifier, list-directed I/O is performed, as shown in the following syntax line:

```
WRITE (6, *) iolist
```

In this statement, the asterisk indicates a list-directed transfer. For more information, see Section 3.8, "List-Directed I/O."

3.2.7.6 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 the elements of the array concatenated in column-major order (see the **DIMENSION** entry in Section 4.2, "Statement Directory," for information on order of array elements).

A noncharacter array can also be specified where the elements of the array are treated as equivalent character variables of the same length.

If a Hollerith constant is used to specify an edit list, the edit list cannot contain an apostrophe edit descriptor or another Hollerith constant.

Example

The following example uses a character array, `char`, and two real arrays, `real1` and `real2`, to print a header (`The results are:`) and the values of two integer arrays, three times:

```
$NOTSTRICT
      INTEGER  array1(6), array2(4)
      CHARACTER*8  char(4)
      REAL*8  real1(4), real2(4)
 C    format is   ==>   (" The results are: ", 6I6, 4I6)
      DATA char  /  "(' The r", "esults a", "re: ', 6",
     +              "I6, 4I6)" /
      DATA real1 /  "(' The r", "esults a", "re: ', 6",
     +              "I6, 4I6)" /
      DATA real2 / 8H(' The r, 8Hesults a, 8Hre: ', 6,
     +             8HI6, 4I6)  /
      WRITE (*, char ) array1, array2
      WRITE (*, real1) array1, array2
      WRITE (*, real2) array1, array2
```

3.2.7.7 Character Array Element

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

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

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

3.2.8 Namelist Specifier (NML=)

Within a **WRITE** statement, a namelist specifier causes the names and values of all the variables in the namelist to be written to the specified unit. Within a **READ** statement, a namelist specifier prompts the user to enter a value for one or more of the variables in the namelist. For more information, see Section 3.9, "Namelist-Directed I/O."

3.2.9 File Structure (FORM=)

The structure of a file is determined by the file's access, set by the **ACCESS=** option described in Section 3.2.3, and the form of the data the file contains.

A file is structured in one of three ways:

Form	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 (a carriage-return and line-feed sequence). 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. I/O operations that can be performed on unformatted direct files produce the same results when applied to binary direct files.

See *Microsoft FORTRAN Advanced Topics* for information on how records are represented on your system.

3.2.10 Input/Output List

The input/output list, *iolist*, specifies the items to be input or output. See Section 3.7.3 for an explanation of how the *iolist* and *editlist* interact.

The following items can be in an *iolist*:

- No entry

  An *iolist* can be empty. The resulting record is either of zero length or contains only padding characters.

```
C       An empty iolist:
        WRITE (unit, '(10I8)')
```

- A variable name, an array-element name, a structure-element name, or a character-substring name

  These elements specify that the variable, array element, structure element, or character substring should be input or output.

```
C       A variable and array element in iolist:
        READ (*, 500) n, bahb(n)
```

- An array name

  An unsubscripted array name specifies in column-major order all the elements of the array. See the **DIMENSION** entry in Section 4.2, "Statement Directory," for an explanation of how arrays are arranged in memory.

```
C       An array in the iolist:
        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, either numeric, logical, or character.

- An implied-**DO** list

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

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

  Here, *iolist* is an input/output list (which 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 the **DATA** entry in Section 4.2, "Statement Directory."

An implied-**DO** list is analogous to an ordinary **DO** loop. The *start*, *stop*, and *inc* parameters determine the number of iterations, and *dovar* (where appropriate) can be used as an array element specifier.

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

```
C     Input and output with an implied-do list in iolist:
      INTEGER c, handle(10), n
      WRITE  (*, *) ' Enter c (<=10) and'
      WRITE  (*, *) ' handle(1) to handle(c)'
      READ   (*, 66) c, (handle(n), n = 1, c)
66    FORMAT (I5, 10(:, /, I5))
      WRITE  (*, 99) c, (handle(n), n = 1, c)
99    FORMAT (1X, 2I5)
      END
```

3.2.11 File Sharing (MODE=, SHARE=)

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

Option	Specification
MODE=*mode*	How the first process to open a file can use that file. The file can be opened for reading (**'READ'**), writing (**'WRITE'**), or both (**'READWRITE'**).
SHARE=*share*	How subsequent processes are allowed to access the file (while that file is still open). Subsequent processes can be allowed to read (**'DENYWR'**), to write (**'DENYRD'**), to both read and write (**'DENYNONE'**), or to neither read nor write (**'DENYRW'**). In addition, all processes (except the process that originally opened the file) can be prohibited from opening the file (**'COMPAT'**).

Table 3.4, below, indicates the restrictions on opening a file that has already been opened with a particular value of *mode* and *share*.

Table 3.4 Mode and Share Values

Original Process Opened with MODE=	Original Process Opened with SHARE=	Concurrent Processes Can Be Opened with MODE=	Concurrent Processes Can Be Opened with 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 opened concurrently	
'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, a file 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, 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 the example above, you want the original process to read the file only. Therefore, the original process should open the file with **MODE='READ'**. Next, 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. The first process should open the file with **SHARE='DENYWR'**. Now, as indicated in Table 3.4, other processes can also open the same file with **MODE='READ'** and **SHARE='DENYWR'**.

3.2.12 Record Number (REC=)

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

3.3 Choosing File Types

The available file properties can be combined to create many kinds of files. Two common file types are listed below:

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

   When reading from the asterisk unit (the keyboard), lines must be terminated 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, each 7
C     columns wide, from a file whose name is entered by
C     the user, to another file named OUT.TXT, reversing
C     the positions of the first and second columns.

      PROGRAM  ColSwp
      CHARACTER*64  fname

C     prompt for and read the file name:
      WRITE  (*, 900)
900   FORMAT (' enter FILE NAME - ' \)
      READ   (*, '(A)') fname

C     use unit 3 for input from external file:
      OPEN   (3, FILE = fname)
```

```
C     use unit 4 for output to second external file:
      OPEN    (4, FILE = 'OUT.TXT')

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   (*, '(A)') 'Done'
      END
```

The file type you choose depends on your application. The following is a list of situations when file types other than formatted/sequential are used:

Situation	File-Type Consideration
You need random access I/O	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 is 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 carriage-return/line-feed) instead of passing them through to the program unaltered. The binary format is preferable, since the file contains no record-structure information.
Data must be transferred without any system interpretation, and will be read by non-FORTRAN programs	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. Opening a file as **FORM=BINARY** and **ACCESS=DIRECT** with **RECL=1** lets you move to any position and read an arbitrary sequence of values. Incomplete records don't cause undefined values because the record size is 1.

3.4 File Position

Opening a sequential file positions the file at its beginning (unless the file was opened with **ACCESS = 'APPEND'**). If you write to the file, all records after the current record are discarded. The file position after sequential **WRITE** statements is at, but not beyond, the end-of-file record.

Executing the **ENDFILE** statement inserts an end-of-file marker at the file's current position, then positions the file after the end-of-file marker. A **BACKSPACE** statement must be executed to position the file in front of the end-of-file marker. Any data past this position is lost.

Executing a **READ** statement at the end of the file positions the file beyond the end-of-file record and produces an error unless you have specified the **END=**, **ERR=**, or **IOSTAT=** option in a **READ** statement.

3.5 Internal Files

An external file is a physical device, such as a printer or a screen, or is a file that is known to the operating system. An internal file is a character variable, character array, character array element, character structure element, character substring, or noncharacter array.

There are two basic types of internal files:

Type	Properties
Character variable, character array element, character structure 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.

You must follow these rules when using internal files:

1. Use only formatted, sequential I/O.
2. Use only the **READ** and **WRITE** statements.
3. Don't use list-directed formatting.

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 FORTRAN internal memory representations. That is, reading from an internal file converts the ASCII values into numeric, logical, or character values, and writing to an internal file converts values into their ASCII representations.

If less than an entire record is written to an internal file, the rest of the record is filled with blanks.

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

The following example prompts for a 3-digit file identification 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), '(I3.3)') id
OPEN  (1, FILE = fname, STATUS = 'NEW')
.
.
.
END
```

3.6 Carriage Control

When formatted I/O is used to transfer a record to a terminal device, such as the screen or a printer, 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 3.5 below. 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 example above produces the following output:

```
5 years
```

In list-directed I/O, the first character is not treated as a carriage-control character. In the following example of unformatted I/O, the full string is displayed:

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

The following output is produced:

```
25 years
```

The effects of the characters 0, 1, +, and the blank character are listed in Table 3.5.

Table 3.5 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.[1]
+	Does not advance. Permits overprinting.

[1] 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 graphics character on the screen. The effect is that the character is ignored.

When writing to the screen or printer, the end-of-record mark (a carriage-return and line-feed sequence) is normally not sent until the next record is written. However, if a write to the screen is followed by a read 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 user's input on the same line as the previous output, add the backslash (\) edit descriptor at the end of the **WRITE** statement's edit list. The input then appears at the end of the last line written. Since input lines are always terminated by the user 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 operation.

3.7 Formatted I/O

If a **READ** or **WRITE** statement includes a format specifier (other than an asterisk), the I/O statement is called 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 3.2.7 for information on format specifiers.

3.7.1 Nonrepeatable Edit Descriptors

Table 3.6 summarizes the nonrepeatable edit descriptors. A discussion of edit descriptors follows.

Table 3.6 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
:	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 3.7.1.1–3.7.1.10 describe each nonrepeatable edit descriptor.

3.7.1.1 Apostrophe Editing

If a format specifier contains a character constant, *string*, that *string* is transmitted to the output unit. Embedded blanks are significant.

Two adjacent apostrophes must be used to represent a single apostrophe within a character constant. Each additional level of nested apostrophes requires twice as many apostrophes as the previous level to resolve the ambiguity of the apostrophe's meaning. Note how in the second **WRITE** statement in the example below, the set of apostrophes that delimit the output string within the **FORMAT** statement are doubled, and four apostrophes are required within the output string itself to specify a single output apostrophe.

Example

```
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'
```

Apostrophe editing cannot be used with **READ** statements.

3.7.1.2 Hollerith Editing (H)

The *n***H** edit descriptor transmits the next *n* characters, including blanks, to the output unit. Hollerith editing can be used in every context where character constants can be used.

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

Example

```
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)
```

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

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 new position 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 absolute tabbing; the transmission of the next character is to occur at the character position *c*. The **TR***c* edit descriptor specifies relative tabbing to the right; the transmission of the next character is to occur *c* characters beyond the current position. The **TL***c* edit descriptor specifies relative tabbing to the left; 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 current record, **TL***c* editing causes transmission to or from position 1.

Left tabbing is legal within records written to devices. However, if the record that is written is longer than the buffer associated with the device, you cannot left-tab to a position corresponding to the previous buffer.

For example, the buffer associated with the console is 132 bytes. If a record of 140 bytes is written to the console, left tabbing is allowed for only eight bytes, since the first 132 bytes of the record have been sent to the device and are no longer accessible.

If one of these edit descriptors is used to move to a position to the right of the last data item transmitted and another data item is then written, the space between the previous end of data in the record and the new position is filled with spaces. For example, consider the following:

```
      WRITE (*, '('' '', 3(''1234567890''))')
      WRITE (*, 100) 5, 9
  100 FORMAT (I5, 20X, TL10, I5)
```

This example produces the following output:

```
12345678901234567890123456789 0
    5                9
```

Be careful when using these edit descriptors if you read data from files that use commas as field delimiters. If you move backwards in a record using **TL***c* or **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 occurs. If you want to move backward in a record without disabling commas as field delimiters, advance to the end-of-record mark, then use the **BACKSPACE** statement to move to the beginning of the record.

3.7.1.4 Positional Editing (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.

Example

```
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  (*, 200)
200   FORMAT (1X, 'zogozogozogo', TL8, 'ZOG', 10X, '!')
```

3.7.1.5 Optional-Plus Editing (SP, SS, S)

The **SP**, **SS**, and **S** edit descriptors 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.

Example

```
C     The following statements write: 251 +251  251 +251
C     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  (*, 200) r, r, r, r, r
200   FORMAT (1X, E8.3E1, SP, E8.3E1, SS, E8.3E1, SP,
     +            E8.3E1, S,  E8.3E1)
```

3.7.1.6 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 at the beginning of the next record.

Example

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

The output from this example is shown below:

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

3.7.1.7 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.

When the format controller terminates a transmission to the terminal, it writes an end-of-record mark (a carriage-return and line-feed sequence). If the last edit descriptor encountered by the format controller is a backslash (\), no end-of-record mark is written, so the next I/O statement will continue writing on the same line.

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 \)') 'enter an integer --> '
      READ  (*, '(BN, I6)') j
```

3.7.1.8 Terminating Format Control (:)

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

Example

```
C     The following example writes 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)
      END
```

3.7.1.9 Scale-Factor Editing (P)

The *k*P edit descriptor sets the scale factor for all subsequent F and E edit descriptors (for information on F and E, see Section 3.7.2, "Repeatable Edit Descriptors") until another *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 there is no explicit exponent, the value read in is divided by 10^k before it is assigned to a variable. If there is an explicit exponent, the scale factor has no effect.
- On output, with **F** editing, the value to be written out is multiplied by 10^k before it is displayed.
- On output, with **E** editing, the real part of the value to be displayed is multiplied by 10^k, and its exponent is reduced by k. This alters the column position of the decimal point but not the value of the number.

Examples

The following 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)
```

Assume the following data is 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 example 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)
```

This program outputs the following:

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

3.7.1.10 Blank Interpretation (BN, BZ)

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

The **BN** edit descriptor ignores blanks; it takes all the nonblank characters in the field and right-justifies them. For example, if an input field formatted for six integers contains `' 23 4 '`, it is interpreted as `'  234'`.

The **BZ** edit descriptor makes blanks identical to zeros. Trailing blanks in the field become zeros. Interspersed blanks also become zeros. In the previous example, the input field `' 23 4 '` would be interpreted as `' 23040'`. If `' 23 4'` were entered, the formatter would add one blank to pad the input to the six-integer format, but this extra space would be ignored, and the input would be interpreted as `' 2304 '`. Note that the blanks following the **E** or **D** in real-number input are ignored, whatever form of blank interpretation is in effect.

The default, **BN**, is set at the beginning 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) n
100   FORMAT (BN, I6)
```

If you enter any one of the following three records and terminate by pressing ENTER, the **READ** statement interprets that record as the value `123`:

```
   123
123
123   456
```

Because the repeatable edit descriptor associated with the I/O list item `n` 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). 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)') n
```

The I/O system is looking for five characters that it will try to interpret as an integer number. You have only entered three, so the first thing the I/O system does is to pad the record `502` on the right with two blanks. If **BZ** editing were in effect, those two blanks would be interpreted as zeros, and the record would be equal to `50200`. However, with **BN** editing in effect (the default), the nonblank characters (`502`) are right-justified, so the record is equal to `502`.

3.7.2 Repeatable Edit Descriptors

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

- On input, fields that are all blanks are always interpreted as zero. The interpretation of trailing and interspersed blanks is controlled by the **BN** and **BZ** editing descriptors. Plus signs are optional. The blanks supplied by the file system to pad a record to the required size are not significant.
- On input with **F**, **E**, **G**, and **D** editing, an explicit decimal point in the input field overrides any 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 if 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. The next field starts at the character following the comma. The missing characters are not significant. For example, consider the following **READ** statement:

  ```
  READ (*, '(BZ 3I5)') i, j, k
  ```

  Entering the following data results in `i = 1`, `j = 20`, and `k = 3`:

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

  Do not use this feature when you use explicit positional editing (the **T**, **TL**, **TR**, or *n***X** edit descriptors).
- Two successively interpreted edit descriptors of the types **F**, **E**, **G**, and **D** are required to format complex numbers. Two different descriptors may be used. The first edit descriptor specifies the real part of the complex number, and the second specifies the imaginary part.
- Nonrepeatable edit descriptors may appear between repeatable edit descriptors.

The following sections describe each repeatable edit descriptor.

3.7.2.1 Integer Editing (I)

Syntax

I*w*[[**.***m*]]

On input, any value entered that is associated with an **I** edit descriptor must have the form of an integer (it may not contain a decimal point or exponent), or a run-time error occurs. On output, the I/O list item associated with an **I** edit descriptor must have an integer value, or a run-time error occurs.

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)') 7, 7
```

The following output is produced:

```
7   007
```

3.7.2.2 Hexadecimal Editing (Z)

Syntax

Z[[*w*]]

Hexadecimal editing converts between external data in hexadecimal form (the hexadecimal digits 0 through 9 and A through F) and internal four-bit binary data (0000 through 1111). Each byte of internal data corresponds to two four-bit hexadecimal characters. For example, the ASCII character "m," which is 01101101 in binary, is output as the hexadecimal characters 6D. Similarly, an **INTEGER*4** value is output in its eight-hexadecimal-character form.

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*2** value is represented by four hexadecimal characters.

On output, character data types are written in the same order they appear in memory. For numeric and logical types, bytes are output in order of significance, 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 bytes in memory on an 8086-based machine 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 $w >= 2*n$, the rightmost $2*n$ hexadecimal characters are taken from the input field.
	If $w < 2*n$, the first *w* hexadecimal characters are read from the external field. Enough leading zeros are added to make the width equal to $2*n$.

Blanks in an input field are treated as zeros.

Hexadecimal editing differs from conventional decimal editing in two significant ways. If there are more hex characters to be output than the field width can accommodate, the field is not filled with asterisks. Instead, *w* right-hand characters are displayed.

The left-hand padding of short input fields does *not* take into account the sign bit of the entered value. For example, in a 8-character input field where only `FFFF` is entered, the entry is interpreted as 65,535, not –1.

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

Examples

The following example demonstrates hexadecimal editing for output:

```
      CHARACTER*2 alpha
      INTEGER*2   num

      alpha = 'YZ'
      num   =  4096

      WRITE (*, '(1X, Z, 1X, Z2, 1X, Z6)') alpha, alpha,
     +alpha
      WRITE (*, '(1X, Z, 1X, Z2, 1X, Z6)') num,   num,   num
```

This example produces the following output:

```
595A 5A 00595A
1000 00 001000
```

As 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
`Z`	`YZ`
`Z2`	`0Y`
`Z6`	`YZ`

3.7.2.3 Real Editing without Exponents (F)

Syntax

F*w.d*

On output, the I/O list item associated with an **F** edit descriptor must be a single- or double-precision number, or a run-time error occurs. On input, the number entered may have any integer or real form, as long as its value is within the range of the associated variable.

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). Appearing 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 (*, '(F8.3)') xnum
```

The above statement reads a given input record as follows:

Input	Number Read
`5`	`.005`
`2468`	`2.468`
`-24680`	`-24.680`

```
-246801          -246.801
56789            5.678
-28E2            -2.800
```

The output field occupies *w* characters. One character is a decimal point, leaving *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

```
      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)
```

The program above produces the following output:

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

3.7.2.4 Real Editing with Exponents (E)

Syntax

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

On output, the I/O list item associated with an **E** edit descriptor must be a single- or double-precision number, or a run-time error occurs. On input, the number entered may have any integer or real form, as long as its value is within the range of the associated variable.

The field is *w* characters wide. The *e* parameter is ignored in input statements. 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 3.7.

Table 3.7 Forms of Exponents for the 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 $(0<k<d+2)$, 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.

3.7.2.5 Real Editing–Wide Range (G)

Syntax

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

On output, the I/O list item associated with a **G** edit descriptor must be a single- or double-precision number, or a run-time error occurs. On input, the number entered may have any integer or real form, as long as its value is within the range of the associated variable.

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 3.8 and 3.9 show how the **G** edit descriptor is interpreted.

Table 3.8 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 3.9 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*

3.7.2.6 Double-Precision Real Editing (D)

Syntax

D*w.d*

On output, the I/O list item associated with a **D** edit descriptor must be a single- or double-precision number, or a run-time error occurs. On input, the number entered may have any integer or real form, as long as its value is within the range of the associated variable. All parameters and rules for the **E** descriptor apply to the **D** 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 3.10.

Table 3.10 Forms of Exponents for the 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.

3.7.2.7 Logical Editing (L)

Syntax

L*w*

The field is *w* characters wide. On output, the *iolist* element associated with an **L** edit descriptor must be of type logical or a run-time error occurs. 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 also valid inputs. On output, $w-1$ blanks are followed by either **T** or **F**, as appropriate.

3.7.2.8 Character Editing (A)

Syntax

A[[*w*]]

If *w* is omitted, the field width defaults to the number of characters in the *iolist* associated item. The *iolist* item may be of any type. If it is not of the **CHARACTER** type, 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.

If the number of characters input is less than *w*, the input field is padded with blanks. If the number of characters input is greater than *w*, the input field is truncated on the right to the length of *w*. Only after these adjustments have been made will the input field be transmitted to the *iolist* item. For example, look at the following program fragment:

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

Assume the following 13 characters are typed in at the keyboard:

```
ABCDEFGHIJKLM
```

The following two steps occur:

1. Blanks are added to pad the input field to 15 characters:

   ```
   'ABCDEFGHIJKLM  '
   ```

2. The rightmost 10 characters are transmitted to the *iolist* element `char`:

   ```
   '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.

3.7.3 Interaction between Format and I/O List

If an *iolist* contains one or more items, 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*. A formatted **WRITE** statement with an empty edit specification writes a carriage return and line feed. 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 is less than the number of characters specified by the edit descriptors and *iolist*, the following occurs:

1. The record is padded with blanks on the right to the required length.
2. Any blanks entered by the user are interpreted according to the blank-editing descriptor in effect (**BN** or **BZ**).

For example, consider the following **READ** statement that uses **BZ** editing:

```
READ (*, '(BZ, I5)') n
```

Assume you enter the following in response:

```
5
```

The total number of characters in the input record is two (a blank followed by a `5`). The record is padded on the right with three blanks, but these additional blanks added by the formatter are ignored. The input record is thus interpreted as 5, rather than 5000.

Each item in the *iolist* is associated with a repeatable edit descriptor during the I/O statement execution. Each **COMPLEX** item in the *iolist* requires two edit descriptors in the **FORMAT** statement or format descriptor. Nonrepeatable edit descriptors are not associated with items in the *iolist.*

During the formatted I/O process, the format controller scans and processes the format items from left to right. Following is a list detailing situations the format controller may encounter and their explanations:

- A repeatable edit descriptor is encountered, 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 is encountered, and no corresponding item appears in the *iolist.*

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

```
      i = 5
      WRITE (*, 100) i
100   FORMAT (1X, 'I= ', I5, ' J= ',
     +I5, ' K= ', I5)
```

  the output would look like this:

```
I=      5 J=
```

  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 is encountered, and there are no further items in the *iolist*.

  The format controller terminates I/O.

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

  The format controller terminates I/O.

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

  The colon edit descriptor is ignored.

- The matching final right parenthesis of the format specification is encountered, and there are further items in the *iolist*.

  The file is positioned at the beginning of the next record and the format controller starts over 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, any remaining characters of the record are ignored. When the format controller terminates on output, an end-of-record mark is written.

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.

3.8 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.*

3.8.1 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 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 input list item.
- Blanks are always treated as separators and never as zeros.
- Embedded blanks can only appear within character constants, as specified in the list below.

  Note that the end-of-record mark has the same effect as 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 (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 apostrophes. Each apostrophe within a character constant delimited by apostrophes must be represented by two apostrophes, 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 current definition 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

The following 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
END
```

This program produces the following output:

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

3.8.2 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 length 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). To provide carriage control when the record is printed, each output record automatically begins with a blank 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 in the format of an **I11** edit descriptor.	
Single- and double-precision real constants	Output in the format 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. They are neither preceded nor followed by a value separator. Each internal apostrophe 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

The following 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
END
```

The example above produces the following output:

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

3.9 Namelist-Directed I/O

Namelist-directed I/O is a powerful method for reading data in or writing data out to a file (or the terminal). By specifying one or more variables in a namelist group, you can read or write the values of all of them with a single I/O statement.

A namelist-directed input statement scans the input file for the group name. Once found, the statement then scans for assignment statements that give values to one or more of the variables in the group. Namelist-directed input is terminated with a slash. A namelist-directed output statement writes the name of the namelist, followed by the name of each variable in the namelist, an equal sign, and the variable's current value. Namelist-directed output is terminated with a slash.

A namelist group is created with the **NAMELIST** statement. It takes the form:

NAMELIST / *namelist* / *variablelist*

where *namelist* is an identifying name for the group, and *variablelist* is a list of variables and array names.

The values of the namelist variables are written to a file or the screen with a **WRITE** statement in which *namelist* appears instead of a format specifier. Note that no *iolist* is needed or permitted.

WRITE (*, NML = *namelist*)

NML= is optional, and is only required if other keywords (such as **END=**) are used.

The first output record is an ampersand (&), immediately followed by the namelist group name, in uppercase. Succeeding records list all variable names in the group and their values. Each output record begins with a blank character to provide carriage control if the record is printed. Values take the output format they would have in list-directed I/O, with one exception: character strings are delimited by apostrophes. This permits the file created to be read by a namelist-directed **READ** statement, which requires apostrophes as string delimiters. The last output record is a slash.

Example

The following example declares a number of variables which are placed in a namelist, initialized, then written to the screen with namelist I/O:

```
INTEGER    int1*1, int2*2, int4*4, array(3)
LOGICAL    log1*1, log2*2, log4*4
REAL       real4*4, real8*8
COMPLEX    z8*8, z16*16
CHARACTER  char1*1, char10*10

NAMELIST /example/ int1, int2, int4, log1, log2, log4,
+          real4, real8, z8, z16, char1, char10, array

int1     =  11
int2     =  12
int4     =  14
log1     = .TRUE.
log2     = .TRUE.
log4     = .TRUE.
real4    =  24.0
real8    =  28.0d0
z8       = (38.0,0.0)
z16      = (316.0d0,0.0d0)
char1    = 'A'
char10   = '0123456789'
array(1) = 41
array(2) = 42
array(3) = 43

WRITE (*, example)
```

Output

```
&EXAMPLE
INT1 =          11
INT2 =          12
INT4 =          14
LOG1 = T
LOG2 = T
LOG4 = T
REAL4 =      24.000000
REAL8 =      28.000000000000000
Z8 =            (38.000000,0.000000E+00)
Z16 =         (316.000000000000000,0.000000000000000E+000)
CHAR1 = 'A'
CHAR10 = '0123456789'
ARRAY =             41          42          43
/
```

The operation of a namelist-directed **READ** statement is almost the reverse of a **WRITE** operation. The statement first scans the file (either at the terminal or on disk) from its current position until it finds an ampersand immediately followed by the namelist group name, or until it reaches the end of the file. (Ampersands followed by other names are ignored.) There must be at least one blank or carriage return following the group name to separate it from the following value-assignment pairs.

A value-assignment pair (optional) consists of a variable name, array element, or substring followed by an equal sign and one or more values and value separators. The equal sign may be preceded or followed by any number of blanks (including no blanks). A value separator is a single comma or one or more blanks. A comma that is not preceded by a value is treated as a null value, and the corresponding variable or array element is not altered.

Variables may appear in any order. The same variable may appear in more than one assignment pair. Its final value is the value it received in its last assignment. All the variables in a namelist do not need to be assigned values; those which do not appear, or which are associated with null values, keep their current values. A variable name in the input file that is not in the namelist group causes a run-time error.

If an array name appears without a qualifying subscript, the first value in the assignment statement is given to the first array element, the second to the second element, and so forth. Assignment to arrays is by row-major order.

You may not assign more values than there are elements in an array. For example, you may not specify 101 values for a 100-element array. However, an array need not have values assigned to all its elements. Any missing values are treated as nulls, and the corresponding array elements are not altered. Individual values may also be assigned to subscripted array elements.

A value may be repeated by placing a repeat factor and an asterisk in front of the value. For example, `7*'Hello'` assigns `'Hello'` to the next seven

elements in an array or variable list. A repeat factor and asterisk without a value indicates multiple null values. The corresponding variables are not altered. Given the array `matrix(0:101)`, the following statements assign 10 to element 0, assign 25 to the elements 1 through 50, leave 51 through 100 alone, assign –101 to element 101, then change the value of matrix(42) to 63:

```
matrix = 10, 50*25, 50*, -101
matrix(42) = 63
```

Character strings must be delimited by apostrophes.

A namelist-directed **READ** statement is successfully terminated by a slash or by reaching the end of the file (in which case, an error results). Slashes must not be used as value separators unless it is desired to prematurely terminate the read.

Suppose the preceding program wanted to read new values for some of the variables in namelist `example`. If a file connected to unit four contained the following namelist specifier and assignment statements:

```
&example
z8 = (99.0,0.0)
INT1=99
array(1)=99
real4 = 99.0
CHAR1='Z'
char10(5:9) = 'GrUMp'
log1=F
/
```

then the following namelist-directed **READ** statement would assign new values to the specified variables:

```
READ (UNIT = 4, example)
```

A second `WRITE (*, example)` statement would display their changed values as follows:

```
&EXAMPLE
INT1 =          99
INT2 =          12
INT4 =          14
LOG1 = F
LOG2 = T
LOG4 = T
REAL4 =      99.000000
REAL8 =      28.000000000000000
Z8 =            (99.000000,0.000000E+00)
Z16 =          (316.000000000000000,0.000000000000000E+000)
CHAR1 = 'Z'
CHAR10 = '0123GrUMp9'
ARRAY =             99           42           43
/
```

CHAPTER 4

Statements

4

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 the Microsoft FORTRAN statements, listed alphabetically.

A FORTRAN statement consists of an initial line, optionally followed by up to 19 continuation lines. In Microsoft FORTRAN, the number of continuation lines is limited only by available memory. Statements are written in columns 7 through 72. They 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.

4.1 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 describe, classify, or specify the format of program elements, such as entry points, data, or program units.

Table 4.1 summarizes the FORTRAN statements.

Table 4.1 Categories of FORTRAN Statements

Category	Type	Description
Assignment statements	Executable	Assign a value to a variable or an array element. See the **ASSIGN** and Assignment entries in Section 4.2, "Statement Directory," for more information.
BLOCK DATA, **ENTRY**, **FUNCTION**, **INTERFACE TO**, **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 4.3.
DATA statement	Nonexecutable	Assigns initial values to variables.
FORMAT statement	Nonexecutable	Provides data-editing information.
I/O statements	Executable	Transfer data and manipulate files and records. See Table 4.4 and Chapter 3, "The Input/Output System."
Specification statements	Nonexecutable	Define the attributes of variables, arrays, and subprograms. See Table 4.2.
Statement-function statements	Nonexecutable	Define simple, locally used functions.

Table 4.2 summarizes the specification statements.

Table 4.2 Specification Statements

Statement	Purpose
AUTOMATIC	Declares a variable on the stack, rather than at a static memory location
COMMON	Shares variables between two or more program units
DIMENSION	Identifies a variable as an array and specifies the number of 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	Changes the default typing for real and integer variables and functions
INTRINSIC	Allows a predefined function to be passed as an argument
MAP...END MAP	Within a **UNION** statement, delimits a group of variable type declarations that are to be ordered contiguously within memory
NAMELIST	Declares a group name for set of variables to be read or written in a single statement
PARAMETER	Equates a constant expression with a name
RECORD	Declares one or more variables of a user-defined structure type
SAVE	Causes variables to retain their values between invocations of the procedure in which they are defined
STRUCTURE...END STRUCTURE	Defines a new variable type, composed of a collection of other variable types
Type: **CHARACTER**[[**n*]] **COMPLEX**[[*bytes]] **DOUBLE COMPLEX** **DOUBLE PRECISION** **INTEGER**[[*bytes]] **LOGICAL**[[*bytes]] **REAL**[[*bytes]] **RECORD** */struct-name/* **STRUCTURE** */struct-name/*	Specifies the type of user-defined names
UNION...END UNION	Within a **STRUCTURE** statement, causes the variables in two or more maps to occupy the same memory location

Table 4.3 summarizes the control statements.

Table 4.3 Control Statements

Statement	Purpose
ALLOCATE	Dynamically establishes allocatable array dimensions
CALL	Executes a subroutine
CASE	Within a **SELECT CASE** structure, marks a block of statements that are executed if an associated value matches the **SELECT CASE** expression
CONTINUE	Has no effect; often used as target of **GOTO** or as the terminal statement in a **DO** loop
CYCLE	Advances control to the end statement of a **DO** loop; the intervening loop statements are not executed
DEALLOCATE	Frees the storage space previously reserved in an **ALLOCATE** statement
DO	Evaluates statements in the **DO** loop, through and including the ending statement, a specific number of times
DO WHILE	Evaluates statements in the **DO WHILE** loop, through and including the ending statement, until a logical condition becomes false
ELSE	Introduces an **ELSE** block
ELSE IF	Introduces an **ELSE IF** block
END	Ends execution of a program unit
END DO	Marks the end of a series of statements following a **DO** or **DO WHILE** statement
END IF	Marks the end of a series of statements following a block **IF** statement
END SELECT	Marks the end of a **SELECT CASE** statement
EXIT	Leaves a **DO** loop; execution continues with the first statement following
GOTO	Transfers control to another part of the program
INCLUDE	Inserts contents of a specified file into the source file
IF	Controls conditional execution of other statement(s)
PAUSE	Suspends program execution and, optionally, executes operating-system commands
RETURN	Returns control to the program unit that called a subroutine or function
SELECT CASE	Transfers program control to a selected block of statements, based on value of a controlling expression
STOP	Terminates a program

Table 4.4 summarizes the I/O statements.

Table 4.4 I/O Statements

Statement	Purpose
BACKSPACE	Positions a file to the beginning of the previous record
CLOSE	Disconnects the specified unit
ENDFILE	Writes an end-of-file record
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 file
PRINT	Displays data on the screen
READ	Transfers data from a file to the items in an I/O list
REWIND	Repositions a file to its first record
WRITE	Transfers data from the items in an I/O list to a file

4.2 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	Statement syntax, and description of statement 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.

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 2.1, "Lines." If the **$FREEFORM** metacommand is specified, follow the formatting rules given in Section 2.3, "Free-Form Source Code." The program fragment below, 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)
```

ACTION

Dynamically sizes an array that has previously been declared with the **ALLOCATABLE** attribute

SYNTAX

ALLOCATE (*array*([[*l*:]]*u*[[,[[*l*:]]*u* ...]]) [[, **STAT** = *ierr*]]) ...

Parameter	Description
array	Name of allocatable array
ierr	Integer variable that returns status of allocation attempt
l	Integer expression that sets the lower bound of the array
u	Integer expression that sets the upper bound of the array

REMARKS

Allocatable arrays may be dynamically allocated and deallocated at run time. An array must have previously been declared **ALLOCATABLE**, and the number of its dimensions declared (with colons only; no bounds may be specified). The **ALLOCATE** statement establishes the upper and lower bounds of each dimension and reserves sufficient memory.

More than one allocatable array name may appear in an **ALLOCATE** statement, separated by commas. The **STAT=** parameter must appear last.

Allocatable arrays may not have the **NEAR** attribute. If the array is to be larger than 65,536 bytes, you must specify the **HUGE** attribute so the array elements are correctly addressed. Allocatable arrays may not appear in **AUTOMATIC**, **COMMON**, **DATA**, **EQUIVALENCE**, or **STRUCTURE** statements.

Attempting to reallocate a previously allocated array causes a run-time error.

Any allocation failure causes a run-time error, unless the **STAT=** option is present. The *ierr* variable returns a value of zero if the allocation was successful, and the number of the run-time error if the allocation failed.

EXAMPLE

```
      .
      .
      .
      INTEGER  dataset[ALLOCATABLE](:,:),
     +         results[ALLOCATABLE, HUGE](:,:,:)
      INTEGER  reactor, level, calcs, error
      DATA  reactor, level, calcs  / 10, 50, 100 /

      ALLOCATE (dataset(reactor,level),
     +          results(reactor,level,calcs), STAT = error)

      IF (error .NE. 0)
     +    STOP 'Not enough storage for data; aborting...'
      .
      .
      .
```

SEE ALSO **DEALLOCATE**

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

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

Parameter	Description
label	A format label or statement label. The *label* must appear in the same program unit.
variable	An integer variable.

REMARKS Variables with label values are used in the following situations:

Situation	Use
An assigned **GOTO** statement	The assigned **GOTO** statement requires a variable with the value of the label of an executable statement.
A format specifier	Input/output statements can accept a variable which specifies 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 example 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 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 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 statement
      GOTO ivar
      WRITE (*, *)' This is never written'
  200 CONTINUE
      WRITE (*, *)' This is written'
  100 FORMAT (' This is format 100')
      END
```

ACTION

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

SYNTAX

variable = *expression*

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

REMARKS

The variable, array, array-element, or structure-element type and the expression type must be compatible:

- If *expression* is numeric, then *variable* must be numeric, and the statement is 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 1.7.1.2, "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 a logical assignment statement.

  Logical *expressions* of any byte size can be assigned to logical *variables* of any byte size without changing 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 a character assignment statement. Both *variable* and *expression* must have type **CHARACTER**. 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.

  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.
  - If *variable* is an array, then array assignment occurs as described in Section 1.7.5, "Array Expressions."

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.'
      END IF
      END
```

The program above has the following output:

```
a =  .1000    b =  .0001
a > b is true.
```

The following fragment demonstrates legal and illegal assignment statements:

```
      INTEGER  i, int
      REAL  rone(4), rtwo(4), x, y
      COMPLEX  z       CHARACTER  char6*6, char8*8

      i       =  4
      x       =  2.0
      z       = (3.0, 4.0)
      rone(1) =  4.0
      rone(2) =  3.0
      rone(3) =  2.0
      rone(4) =  1.0
      char8   = 'Hello,'
```

```
C     The following assignment statements are legal:
      i     = rone(2)
      int   = rone(i)
      int   = x
      y     = x
      y     = z
      y     = rone(3)
      rtwo  = rone
      rtwo  = 4.7
      char6 = char8

C     The following assignment statements are illegal:
      char6 = x + 1.0
      int   = char8//'test'
      y     = rone
```

ACTION Declares specified variables to be on the stack, rather than at a static memory location

SYNTAX **AUTOMATIC** [[*names*]]

Parameter	Description
names	A list of variables or array names to be made automatic. If there is more than one variable, they must be separated by commas.

REMARKS

In Microsoft FORTRAN, all variables are static by default. A variable declared as "automatic" has no fixed memory location; a section of stack memory is allocated for the variable as needed. Automatic variables within procedures are discarded when the procedure completes execution. Therefore, such variables cannot be guaranteed to have the same value on the next invocation of the procedure.

If an **AUTOMATIC** statement contains no variable names, all the variables within that program unit (the main program, or an individual subprogram) that can legally be automatic are implicitly automatic.

Common-block names and variables are not allowed in an **AUTOMATIC** statement. A variable cannot appear in both a **SAVE** statement and an **AUTOMATIC** statement.

Variables with the **ALLOCATABLE**, **EXTERNAL**, **FAR**, or **HUGE** attribute cannot be automatic. A variable that has been explicitly declared automatic may not appear in a **DATA** statement. Variables that are implicitly automatic and appear in a **DATA** statement will be initialized and placed in static memory. A variable may appear in an **AUTOMATIC** statement only once. Formal arguments and procedure names may not appear in an **AUTOMATIC** statement.

The ability to declare automatic variables has been added to support OS/2 multithread applications.

EXAMPLE

```
C     In this example, all variables within the program unit
C     are automatic, except for "clark" and "lois"; these are
C     explicitly declared in a SAVE statement, and thus have
C     static memory locations:

      INTEGER FUNCTION  Fibonacci (clark, lois)
      AUTOMATIC
      SAVE  clark, lois
```

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*]])}

If **UNIT**= is omitted, *unitspec* must be the first parameter. The parameters can otherwise appear in any order.

Parameter	Description
unitspec	An integer expression that specifies an external unit. If *unitspec* is not open, a run-time error occurs.
errlabel	The label of an executable statement in the same program unit. If *errlabel* is specified, an I/O error causes transfer of control to the statement at *errlabel*. If *errlabel* is omitted, the effect of an I/O error is determined by the presence or absence of *iocheck*.
iocheck	An integer variable, array element, or structure element that returns a value of zero if no error occurs, or the number of the run-time error message if an error does occur. For more information on error handling, see Section 3.2.6, "Error and End-of-File Handling."

REMARKS

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

Special Case	Result
No preceding record	The file position is not changed
Preceding record is end-of-file record	The file is positioned before the end-of-file record
File position is in middle of 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

```
BACKSPACE 5
BACKSPACE (5)
BACKSPACE lunit
BACKSPACE (UNIT = lunit, ERR = 30, IOSTAT = ios)
```

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	A global symbolic name for the subprogram. 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 it 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 an executable program. Otherwise, the default name will be defined twice, causing 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**, **DATA**, **END**, **DIMENSION**, **EQUIVALENCE**, **IMPLICIT**, **MAP**, **PARAMETER**, **RECORD**, **SAVE**, **STRUCTURE**, **UNION**, and type statements. No executable statements are permitted.
- 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 must be specified in that block-data subprogram, even if not all of the constituents are initialized. This is because the length of the named common block must be the same in all subprograms.

EXAMPLES

```
C     The following block-data subprogram initializes
C     the named common block /greatlakes/:
C
      BLOCK DATA Lakes
      COMMON /greatlakes/ erie, huron, michigan, ontario, superior
      DATA erie, huron, michigan, ontario, superior /1, 2, 3, 4, 5/
      END
C
C     Using the same common block, /greatlakes/, the
C     following block-data subprogram is NOT allowed;
C     not all the members of /greatlakes/ are specified.

      BLOCK DATA GrLaks
      COMMON /greatlakes/ erie, huron, ontario, superior
      DATA erie, huron, ontario, superior /1, 2, 4, 5/
      END
```

ACTION

Specifies the **BYTE** type for user-defined names. This type is equivalent to **INTEGER*1**.

SYNTAX

BYTE *vname* [[[*attrs*]]] [[(*dim*)]] [[/*values*/]] [[, *vname* [[[*attrs*]]] [[(*dim*)]] [[/*values*/]]]] ...

Parameter	Description
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**, **ALLOCATABLE**, **C**, **EXTERN**, **FAR**, **HUGE**, **NEAR**, **PASCAL**, **REFERENCE**, **VALUE**.
dim	A dimension declarator. Specifying *dim* declares *vname* as an array.
values	A list of constants and repeated constants, separated by commas. A repeated constant takes the form *n***constant*, where *n* is a positive integer constant. The */values/* option initializes *vname*. The following statement declares that `num` is of type **BYTE**, and sets `num` equal to `10`: `BYTE num / 10 /`

REMARKS

A **BYTE** 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.

BYTE statements must precede all executable statements.

EXAMPLES

```
BYTE count, matrix(4, 4) / 4*1, 4*2, 4*4, 4*8 /
```

ACTION Invokes a subroutine

SYNTAX **CALL** *sub* [[([[*actuals*]])]]

Parameter	Description
sub	The name of the subroutine to be executed.
actuals	One or more actual arguments. If there is more than one argument, they are separated by commas. Each 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 returns to the calling program or procedure, 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.

A **CALL** statement must contain as many actual arguments as there are formal arguments in the corresponding **SUBROUTINE** statement (unless the **C** and **VARYING** attributes were used in declaring the subroutine).

If a **SUBROUTINE** statement contains no formal arguments, a **CALL** statement referencing that subroutine must not include any actual arguments. However, an empty pair of parentheses can follow *sub*.

Formal arguments and their corresponding actual arguments must have the same data type (except for 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**.

For all arguments passed by reference (see Section 1.6.11, "VALUE," for information on passing arguments by value), the compiler assumes 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.

If a subroutine call appears more than once in a program unit, the compiler checks that the number and types of actual arguments passed are the same in each call (i.e., consistency).

The compiler also checks that the actual arguments used in calls to a subroutine correspond in number and type to its formal arguments (i.e., validity). In order to do this, the **SUBROUTINE** statement (or an **INTERFACE TO** statement defining the subroutine) must appear in the same source file as any calls to it, and it must precede those calls. The compiler will then ensure that all actual arguments agree with the formal arguments. If the actual arguments are not checked in this way, and if they do not agree with the formal arguments, the result of calling a subroutine is unpredictable.

NOTE *When passing integer and logical arguments, you must pay attention to type agreement. The* ***$STORAGE*** *metacommand controls how integer and logical arguments are passed. When the default (****$STORAGE: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* ***$STORAGE: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****.*

Therefore, if an integer or logical formal argument is of a different type than the default storage type, you must convert the actual argument to the same type. Use the ***INT2*** *and* ***INT4*** *intrinsic functions within the actual argument list, as described in Section 5.1.1, "Data-Type Conversion."*

The alternate-return feature lets you specify the statement to which a subroutine should return control. To use the alternate-return feature, follow these steps:

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 (row, column, *100, *200, *500)
```

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 (r, c, *, *, *)
```

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.

   For example, if the statement `RETURN 1` is reached in the program containing the two statements in steps 1 and 2 above, control returns to the statement at label 100 in the calling routine; if `RETURN 2` is reached, control returns to the statement at label 200; and if `RETURN 3` is reached, control returns to the statement at label 500. If a **RETURN** statement without any number is reached (`RETURN`), or a **RETURN**

statement that has a number for which there is no corresponding return label is reached (such as `RETURN 4`, in this example), control returns to the statement following the **CALL** statement in the calling routine.

A "recursive" subroutine is one that calls itself, or calls another subprogram which in turn calls the first subroutine before the first subroutine has completed execution. FORTRAN does not support recursive subroutine calls.

EXAMPLES

```
      .
      .
      .
      IF (ierr .NE. 0)  CALL Error (ierr)
      END
C
      SUBROUTINE Error (ierrno)
      WRITE (*, 200) ierrno
  200 FORMAT (1X, 'error', I5, ' detected')
      END

C     This example illustrates the alternate return feature:
    1 CALL Boomerang (count, *10, j, *20, *30)
      WRITE (*, *) 'normal return'
      GOTO 40
   10 WRITE (*, *) 'returned to 10'
      GOTO 40
   20 WRITE (*, *) 'returned to 20'
      GOTO 40
   30 WRITE (*, *) 'returned to 30'
   40 CONTINUE
      .
      .
      .
      SUBROUTINE Boomerang (i, *, j, *, *)
      IF (i .EQ. 10)  RETURN 1
      IF (i .EQ. 20)  RETURN 2
      IF (i .EQ. 30)  RETURN 3
      RETURN
```

ACTION

Marks the beginning of a block of statements executed if an item in a list of expressions matches the test expression

SYNTAX

CASE {**DEFAULT** | (*expressionlist*)}
statementblock

Parameter	Description
DEFAULT	The keyword indicating that the following statement block is to be executed if none of the expressions in any other **CASE** statements match the test expression in the **SELECT CASE** statement.
expressionlist	A list of values and ranges of values, which must be constant and must match the data type of the test expression in the **SELECT CASE** statement. The values must be of type **INTEGER**, **LOGICAL**, or **CHARACTER*1**. If *testexpr* matches one of the values, the following block of statements is executed.
statementblock	One or more executable statements. The block may also be empty.

REMARKS

The **CASE** statement may only appear within the **SELECT CASE...END CASE** construct.

There are two ways to include values in the *expressionlist*. The first is to give a list of individual values, separated by commas. The second is to specify an inclusive range of values, separated by a colon, such as `5:10`. If the lower bound is omitted (such as `:10`), then all values less than or equal to the upper bound match. If the upper bound is omitted (such as `5:`), then all values greater than or equal to the lower bound match. Ranges of printable ASCII characters may be included, such as `'I':'N'` or `' ':'/'` . Both individual values and ranges of values may be included in the same *expressionlist*. You cannot specify a range of values when *testexpr* is of type **LOGICAL**. A given value (even when specified implicitly as part of a range) can only appear in one *expressionlist*.

A *statementblock* need not contain executable statements. Empty blocks can be used to make it clear that no action is to be taken for a particular set of expression values.

The **CASE DEFAULT** statement is optional. You can include only one **CASE DEFAULT** statement in a **SELECT CASE** block.

If the value of *testexpr* does not match any value in any *expressionlist*, execution passes beyond the **SELECT CASE** construct to the next executable statement.

EXAMPLE

```
      CHARACTER*1  cmdchar

      SELECT CASE (cmdchar)
      CASE ('0')
      WRITE (*, *) "Must retrieve one to nine files"
      CASE ('1':'9')
      CALL RetrieveNumFiles (cmdchar)
      CASE ('A', 'a')
      CALL AddEntry
      CASE ('D', 'd')
      CALL DeleteEntry
      CASE ('H', 'h')
      CALL Help
      CASE ('R':'T', 'r':'t')
      WRITE (*, *) "REDUCE, SPREAD and TRANSFER commands ",
     +                          "not yet supported"
      CASE DEFAULT
      WRITE (*, *) "Command not recognized; please re-enter"
      END SELECT
```

SEE ALSO **SELECT CASE...END SELECT**

ACTION Specifies the **CHARACTER** type for user-defined names

SYNTAX **CHARACTER**[[**chars*]] *vname* [[[*attrs*]][[**length*]][[(*dim*)]][[/*values*/]]
[[, *vname* [[[*attrs*]][[**length*]][[(*dim*)]]][[/*values*/]]...

The order of the *dim* and *length* parameters can be reversed.

Parameter	Description
chars	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 (*) (see REMARKS, below) in parentheses. The *chars* parameter specifies the length, in characters, 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. The parameter *vname* cannot be the name of a subroutine or main program.
attrs	A list of attributes, separated by commas. The *attrs* describe *vname*. The following attributes can be used with *vname*: **ALIAS, ALLOCATABLE, C, EXTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE.**
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 characters, of the *vname* immediately preceding it. This value, if specified, overrides the length indicated by *chars*.
dim	A dimension declarator. Specifying *dim* 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 integer constant, and is equivalent to *constant* repeated *n* times. The */values/* option, if specified, initializes *vname*. The following statement declares `word` 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 in parentheses ((*)) as a length specifier indicates that the length is specified elsewhere. An asterisk length specifier is allowed in the following cases:

1. Character constants defined by **PARAMETER** statements. The actual length is determined by the length of the character string assigned to the parameter.
2. Formal character arguments. The actual length is determined by the length of the actual argument.
3. Character functions that are defined in one program unit, and referenced in another. The actual length is determined by a type declaration in the program unit that references the function.

If neither *length* nor *chars* is specified, the length defaults to one.

CHARACTER statements must precede all executable statements.

EXAMPLES

```
CHARACTER  wt*10, city*80, ch

CHARACTER  name(10)*20, eman*20(10)

CHARACTER*20  tom, dick, harry*12, tarzan, jane*34
```

ACTION Disconnects a specified unit

SYNTAX

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

If **UNIT**= is omitted, *unitspec* must be the first parameter. The parameters can otherwise appear in any order.

Parameter	Description
unitspec	An integer expression that specifies an external unit. No error occurs if the unit is not open.
errlabel	The label of an executable statement in the same program unit. If *errlabel* is specified, an I/O error causes transfer of control to the statement at *errlabel*. If *errlabel* is omitted, the effect of an I/O error is determined by the presence or absence of *iocheck*.
iocheck	An integer variable, array element, or structure element returns a value of zero if there is no error, or the number of the run-time error if an error occurs. For more information on error handling, see Section 3.2.6, "Error and End-of-File Handling."
status	A character expression that evaluates to either **'KEEP'** or **'DELETE'**. Files opened without a file name are "scratch" files. For these files, the default for *status* is **'DELETE'**. Scratch files are temporary, and are always deleted upon normal program termination; specifying **STATUS='KEEP'** for scratch files causes a run-time error. The default for *status* for all other files is **'KEEP'**. For QuickWin applications, **STATUS='KEEP'** causes the child window to remain on the screen even after the unit closes. The default status is **'DELETE'**, which removes the child window from the screen.

REMARKS Opened files do not have to be explicitly closed. Normal termination of a program will close each file according to its default status.

Closing unit 0 automatically reconnects unit 0 to the keyboard and screen. Closing units 5 and 6 automatically reconnects those units to the keyboard or screen, respectively. Closing the asterisk (*) unit causes 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 operation or the **EOF** intrinsic function to be executed, because the results are unpredictable.

EXAMPLE

```
C     Close and discard file:
      CLOSE (7, STATUS = 'DELETE')
```

ACTION

Allows two or more program units to directly share variables, without having to pass them as arguments

SYNTAX

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

Parameter	Description
cname	A name for the common block. If none is given, the block is said to be "blank common." Omitting *cname* specifies that all elements in the *nlist* that follows are in 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.
nlist	A list of variable names, array names, and array declarators, separated by commas. Formal-argument names, function names, automatic variables, and allocatable arrays 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.

REMARKS

Any common block can appear more than once in the same program unit. The list of elements in a particular common block is treated as a continuation of the list in the previous common block with the same name. For this reason, a given name can appear only once in all the common blocks of a program unit. Consider the following **COMMON** statements:

```
COMMON /ralph/ ed, norton, trixie
COMMON /     / fred, ethel, lucy
COMMON /ralph/ audrey, meadows
COMMON /jerry/ mortimer, tom, mickey
COMMON         melvin, purvis
```

They are equivalent to these **COMMON** statements:

```
COMMON /ralph/ ed, norton, trixie, audrey, meadows
COMMON         fred, ethel, lucy, melvin, purvis
COMMON /jerry/ mortimer, tom, mickey
```

All items in a common block must be only character or only noncharacter items. Microsoft FORTRAN permits both character and noncharacter items in the same common block.

Microsoft FORTRAN normally starts all variables and arrays (including character variables and arrays) on even-byte addresses. If a single (non-array) character variable has an

odd length, the next variable in memory (regardless of type) begins at the next even-byte address.

There are, however, two exceptions. In an array of character variables, each element immediately follows the preceding element, without regard for even or odd addresses. Within a common block, a character variable that follows another character variable always begins on the next available byte. (Noncharacter variables still begin on an even byte.)

This can cause problems if a common block contains odd-length character variables interspersed with noncharacter variables. If the matching common block in another program unit contains only noncharacter variables (in effect, equivalencing the variables), there will be holes in the equivalence due to unused bytes in the block with the odd-length character variables.

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. The blank common block, however, can have a different length in different program units. The blank common block is as long as the longest blank common block in any of the the program units.

A variable that appears in a common block may not be initialized in a **DATA** statement unless that **DATA** statement is part of a block data subprogram. In Microsoft FORTRAN, common-block variables may be initialized in **DATA** statements in any program unit.

An item that appears in *nlist* cannot be initialized in a type statement. The following example causes a compile-time error:

```
INTEGER i /1/
COMMON i
```

EXAMPLE

```
PROGRAM MyProg
COMMON i, j, x, k(10)
COMMON /mycom/ a(3)
.
.
.
END

SUBROUTINE MySub
COMMON pe, mn, z, idum(10)
COMMON /mycom/ a(3)
.
.
.
END
```

ACTION Specifies the **COMPLEX** type for user-defined names

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 symbolic name of a constant, variable, array, external function, statement function, intrinsic function, function subprogram, or array declarator. The *vname* parameter cannot be the name of a subroutine or a main program.
attrs	A list of attributes separated by commas. The *attrs* describe *vname*. The following attributes can be used with *vname*: **ALIAS, ALLOCATABLE, C, EXTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE.**
length	Must be 8 or 16. Specifies length of associated *vname*, in bytes. If *length* is given, it overrides the length specified by *bytes*.
dim	A dimension declarator. Specifying *dim* 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 integer constant, and is equivalent to the constant *constant* repeated *n* times. The */values/* option, if specified, initializes *vname*. The following statement declares that `vector` is of type complex, and sets `vector` equal to `(32.0,10.0)`:

```
COMPLEX vector / (32.0,10.0) /
```

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.

DOUBLE COMPLEX and **COMPLEX*16** represent the same data type.

EXAMPLES

```
COMPLEX  ch, zdif*8, xdif*16

COMPLEX*8  zz

COMPLEX*16  ax, by

COMPLEX  x*16, y(10)*8, z*16(10)
```

ACTION Does not have any effect

SYNTAX **CONTINUE**

REMARKS The **CONTINUE** statement is a convenient place for a statement label, especially as the terminal statement in a **DO** or **DO WHILE** loop.

EXAMPLE

```
      DIMENSION narray(10)
      DO 100, n = 1, 10
         narray(n) = 120
100   CONTINUE
```

ACTION Within a loop, advances control to the terminating statement of a **DO** or **DO WHILE** loop

SYNTAX **CYCLE**

REMARKS The **CYCLE** statement skips over the remaining part of a **DO** or **DO WHILE** loop. By combining a **CYCLE** statement with a logical **IF** statement, you can control whether the subsequent code executes.

EXAMPLE

Suppose you wanted to print a table of relativistic time-dilation factors for every velocity from 0 to the speed of light, in steps of 100 km/second. Perhaps you do not want to calculate these factors for speeds less than 10 percent of the speed of light. The following example computes the time-dilation factors accordingly, putting them in the array `timedilation`. You can use the **WRITE** statement to print out the array.

Time-dilation factor: $1 / \sqrt{1 - (v/c)^2}$

```
INTEGER sub          ! subscript for timedilation array
REAL timedilation(0:300)
speedolight = 300000e3          ! 300000 km per second
speedstep = 100e3               ! 100 km per second

   sub = speedolight / speedstep
DO velocity = 1, speedolight, speedstep
   timedilation(sub) = 1.0
   IF (velocity .LT. (0.1 * speedolight))   CYCLE
   timedilation(sub) =
+  1.0 / SQRT (1.0 - (velocity / speedolight)**2)
END DO
```

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, structure elements, and implied-**DO** lists, separated by commas. Implied-**DO** lists are discussed in the REMARKS section below. Each subscript in *nlist* must be an integer constant expression, except for implied-**DO** variables. Each substring specifier 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 integer constant, and *c* is the constant to be repeated. The repeated constant `3*10`, for example, is equivalent to the *clist* `10,10,10`. A Hollerith constant is written in the form *n***H***data*, where *n* is a positive 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 Structure variables and variables explicitly declared as automatic may not appear in **DATA** statements. However, structure elements may appear in **DATA** statements.

Each noncharacter element in *clist* is converted to the data type of the corresponding element in *nlist*, if necessary.

If a character element in *clist* is shorter than its corresponding variable or array element, it is extended to the length of the variable by adding blank characters to the right. If a character element is longer than its variable or array element, it is truncated. A single character constant defines a single variable or array element. A repeat count can be used.

If the **$STRICT** metacommand is not specified, a character element in *clist* can initialize a variable of any type.

Only local variables, arrays, and array elements can appear in a **DATA** statement. Formal arguments, variables in blank common blocks, and function names cannot be assigned initial values in a **DATA** statement. Variables in named common blocks may not appear in a

DATA statement unless the **DATA** statement is in a block-data subprogram. In Microsoft FORTRAN, elements in named common blocks can be assigned initial values using a **DATA** statement in a main program, function, or subroutine; the **DATA** statement does not have to be in a block-data subprogram.

The form of an implied-**DO** list is as follows:

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

Parameter	Description
dolist	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*	The 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)
```

The number of iterations 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 the **DO** entry in this section for more information. When an implied-**DO** list appears in a **DATA** statement, the list items in *dolist* are initialized once for each iteration of the implied-**DO** list. The range of an implied-**DO** list is *dolist*. 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.

EXAMPLES

```
      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/

C     The following example initializes diagonal and below in
C     a 5x5 matrix:
      DATA  ((x(j,i), i=1,j), j=1,5) / 15*1.0 /
      DATA  pi / 5*3.14159 /
      DATA  list / 100*0 /
      DATA  help(1:4), help(5:8), help(9:12) /3*'HELP'/
```

ACTION Frees the array storage space previously reserved in an **ALLOCATE** statement

SYNTAX **DEALLOCATE** (*arraylist* [[, **STAT** = *ierr*]])

Parameter	Description
arraylist	A list of allocatable array names; if more than one exists, they must be separated by commas.
ierr	The integer variable that returns status of deallocation attempt.

REMARKS The **STAT=** parameter must appear last.

Attempting to deallocate an array that was not allocated causes a run-time error.

Any deallocation failure causes a run-time error, unless the **STAT=** parameter is present. The *ierr* variable returns a value of zero if the deallocation was successful, and the number of the run-time error if the deallocation failed.

If an allocatable array is referenced when it is not currently allocated, the results are unpredictable.

EXAMPLE

```
INTEGER  dataset[ALLOCATABLE](:,:,:)
INTEGER  reactor, level, points, error
DATA  reactor, level, points  / 10, 50, 10 /

ALLOCATE (dataset(1:reactor,1:level,1:points), STAT = error)

DEALLOCATE (dataset, STAT = error)
```

ACTION

Declares that a variable is an array, and specifies the number of dimensions and their bounds

SYNTAX

DIMENSION *array* [[*attrs*]] ({[[*lower*:]]*upper* | :} [[,{[[*lower*:]]*upper* | :}...]])

Parameter	Description
array	The name of an array. More than one array may be declared in a single **DIMENSION** statement. Multiple names are separated by commas.
attrs	A list of attributes separated by commas. The *attrs* describe *array*. The following attributes can be used with *array*: **ALIAS, ALLOCATABLE, 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*.

REMARKS

The specification *array*({[[*lower*:]] *upper* | :}) is called an "array declarator." The [[*lower*:]]*upper* specifier or the **:** specifier is called a "dimension declarator." An array has as many dimensions as it has dimension declarators. You may specify no more than seven dimensions. In Microsoft FORTRAN, the number of dimensions and their sizes are limited only by available memory.

When an array is specified as allocatable, the dimension declarator consists only of a colon for each dimension. The single-colon specifier may be used only when an array is specified as allocatable.

You are free 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 array 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 size is 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 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.
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.

Within noncharacter arrays, all elements begin on even-byte (word) addresses. Within character arrays (and arrays of **INTEGER*1** or **LOGICAL*1** variables), elements always begin at the next available byte (odd or even).

All adjustable- and assumed-size arrays, as well as the bounds for adjustable-size arrays, must be formal arguments to the program unit in which they appear. Allocatable arrays must not be formal arguments.

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, the preceding **DATA** statement produces 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     Array data values are assigned in column-major order
      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',
     +          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)
      END
```

ACTION

Repeatedly executes the statements following the **DO** statement through the statement which marks the end of the loop

SYNTAX

DO [[*label* [[,]]]] *dovar* = *start*, *stop* [[, *inc*]]

Parameter	Description
label	The statement label of an executable statement.
dovar	An integer, real, or double-precision variable, called the **DO** variable.
start, *stop*	The integer, real, or double-precision expressions.
inc	An integer, real, or double-precision expression. The parameter *inc* cannot equal zero; *inc* defaults to one.

REMARKS

The *label* is optional. If *label* is used, the loop terminates with the labeled statement. If *label* is not used, the loop is terminated with an **END DO** statement.

If you use a labeled terminating statement, it must follow the **DO** statement and be in the same program unit. This statement must not be an unconditional or assigned **GOTO**, a block or arithmetic **IF**, **CASE**, **CYCLE**, **DO**, **ELSE**, **ELSE IF**, **END**, **END IF**, **END SELECT CASE**, **EXIT**, **RETURN**, **SELECT CASE**, or **STOP** statement.

The terminal statement may be 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 terminate the **DO** loop unless an alternate-return specifier in a **CALL** statement returns control to a statement outside the **DO** loop.

The following restrictions apply to **DO** loops:

- If a **DO** loop appears within another **DO** or **DO WHILE** loop, its range must be entirely within the range of the enclosing loop.
- If a **DO** statement appears within an **IF**, **ELSE IF**, or **ELSE** block, the **DO** loop must be contained within the block.
- If a block **IF** statement appears within a **DO** loop, its associated **END IF** statement must also be within that **DO** loop.
- The loop variable *dovar* may not be modified by statements within the loop.

- Jumping into a **DO** loop from outside its range is not permitted. However, a special feature added for compatibility with earlier versions of FORTRAN permits extended-range **DO** loops. See the **$DO66** entry in Section 6.2, "Metacommand Directory," for more information.
- Two or more **DO** or **DO WHILE** loops may share one labeled terminal statement. An **END DO** statement may terminate only one **DO** or **DO WHILE** loop.
- If a **SELECT CASE** statement appears within a **DO** loop, its associated **END SELECT CASE** statement must also appear within that **DO** loop.

Note that the number of iterations in a **DO** loop is limited to the maximum possible value of the loop variable. For example, **DO** loops that use **INTEGER*2** variables as the **DO** variable and bounds cannot be evaluated more than 32,767 times. When the **$DEBUG** metacommand is used, a run-time error may occur if a **DO** variable overflows. When **$DEBUG** is not used, the results of an overflow are unpredictable. The following fragment causes an overflow:

```
$DEBUG
      integer*2 i, istart, iend, istep
      data istart /-32000/, iend /32000/, istep /1/
      do 10 i = istart, iend, istep
   10 continue
      end
```

A **DO** statement is executed in the following sequence:

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 calculated using the following formula:

   MAX(INT((*stop* – *start* + *inc*) / *inc*), **0**)

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

   - *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; if not, execution continues with the first statement following the **DO** loop.

   If the **$DO66** metacommand is in effect, the statements in the range of the **DO** loop are executed at least once, regardless of the value of the iteration count.
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 zero, the statements in the range of the **DO** loop are executed again.

NOTE *The iteration count is computed using the integer default size. If the iteration count overflows this precision, the results are unpredictable. The following example causes an iteration overflow:*

```
$STORAGE:2
      IMPLICIT INTEGER*2 (A-Z)
      stop = 32000
      step = 12000
      DO 100  n = 0, stop, step
      WRITE (*, *) 'N = ', n
  100 CONTINUE
```

EXAMPLES

The following two program fragments are examples of **DO** statements:

```
C     Initialize the even elements of a 20-element real array
      DIMENSION array(20)
      DO 100 j = 2, 20, 2
  100 array(j) = 12.0

C     Perform a function 11 times
      DO 200,  k = -30, -60, -3
         int =  j / 3
         isb = -9 - k
         array(isb) = MyFunc (int)
  200 CONTINUE
```

The following shows the final value of a **DO** variable (in this case 11):

```
      DO 200 j = 1, 10
  200 WRITE (*, '(I5)') j
      WRITE (*, '(I5)') j
```

ACTION Executes a block of statements repeatedly while a logical condition remains true

SYNTAX **DO** [[*label* [[,]]]] **WHILE** (*logicalexpr*)

Parameter	Description
label	A label of executable statement or **CONTINUE** statement
logicalexpr	A test expression which evaluates to true or false

REMARKS The *label* is optional. If *label* is used, the loop terminates with the labeled statement. If *label* is not used, the loop is terminated with an **END DO** statement.

A terminating statement must follow the **DO WHILE** statement and be in the same program unit. This statement must not be an unconditional or assigned **GOTO**, a block or arithmetic **IF**, **CASE**, **CYCLE**, **DO**, **ELSE**, **ELSE IF**, **END**, **END IF**, **END SELECT CASE**, **EXIT**, **RETURN**, **SELECT CASE**, or **STOP** statement.

The terminal statement may be a logical **IF** statement.

The range of a **DO WHILE** loop begins with the statement immediately following the **DO WHILE** statement and includes the terminal statement of the **DO WHILE** loop.

Execution of a **CALL** statement that is in the range of a **DO WHILE** loop does not terminate the **DO WHILE** loop unless an alternate-return specifier in a **CALL** statement returns control to a statement outside the **DO WHILE** loop.

The following steps occur when a **DO WHILE** statement is executed:

1. The logical expression is evaluated.
2. If the expression is false, none of the statements within the range of the loop are executed. Execution jumps to the statement following the terminating statement.
3. If the expression is true, the statements within the loop are executed, starting with the first statement following the **DO WHILE** statement.
4. When the terminating statement is reached, execution returns to the **DO WHILE** statement. The logical expression is evaluated, and the cycle repeats.

The following restrictions apply to **DO WHILE** loops:

- If a **DO WHILE** loop appears within another **DO** or **DO WHILE** loop, its range must be entirely within the range of the enclosing loop.
- Two or more **DO** or **DO WHILE** loops may share one labeled terminal statement. An **END DO** statement may terminate only one **DO** or **DO WHILE** loop.
- If a **DO WHILE** statement appears within an **IF**, **ELSE IF**, or **ELSE** block, the range of the **DO WHILE** loop must be entirely within the block.
- If a block **IF** statement appears within a **DO WHILE** loop, its associated **END IF** statement must also be within that **DO WHILE** loop.
- If a **SELECT CASE** statement appears within the range of a **DO WHILE** loop, its associated **END SELECT CASE** statement must also be within that **DO WHILE** loop.
- Jumping into the range of a **DO WHILE** loop is not permitted. However, a special feature added for compatibility with earlier versions of FORTRAN permits extended-range **DO WHILE** loops. See the **$DO66** entry in Section 6.2, "Metacommand Directory," for more information.

EXAMPLE

```
CHARACTER*1 input

input = ' '

DO WHILE ((input .NE. 'n') .AND. (input .NE. 'y'))
WRITE (*, '(A)') 'Enter y or n: '
READ  (*, '(A)') input
END DO
```

ACTION

Specifies the **DOUBLE COMPLEX** type for user-defined names

SYNTAX

DOUBLE COMPLEX *vname* [[[*attrs*]]] [[(*dim*)]] [[/*values*/]]
[[, *vname*[[[*attrs*]]] [[(*dim*)]] [[/*values*/]]]]...

Parameter	Description
vname	The symbolic name of a constant, variable, array, external function, statement function, intrinsic function, **FUNCTION** subprogram, or array declarator. The *vname* parameter cannot be the name of a subroutine or a main program.
attrs	A list of attributes, separated by commas. The *attrs* describe *vname*. The following attributes can be used with *vname*: **ALIAS, ALLOCATABLE, C, EXTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE.**
dim	A dimension declarator. Specifying *dim* 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 integer constant, and is equivalent to the constant *constant* repeated *n* times. The */values/* option, if specified, initializes *vname*. The following statement declares that `vector` is of type **DOUBLE COMPLEX**, and sets `vector` equal to `(32.0,10.0)`:

```
DOUBLE COMPLEX vector / (32.798d2,
+10.985d3) /
```

REMARKS

A **DOUBLE 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 of statement in that program unit.

DOUBLE COMPLEX statements must precede all executable statements.

DOUBLE COMPLEX and **COMPLEX*16** are the same data type.

EXAMPLES

```
DOUBLE COMPLEX  vector, arrays(7,29)

DOUBLE COMPLEX  pi, 2pi /3.141592654,6.283185308/
```

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 symbolic name of a constant, variable, array, external function, statement function, intrinsic function, **FUNCTION** subprogram, or array declarator. The *vname* parameter 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, ALLOCATABLE, C, EXTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE.**
dim	A dimension declarator. Specifying *dim* 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 integer constant, and is equivalent to *constant* repeated *n* times. The */values/* option, if specified, initializes *vname*. The following statement declares that `pi` is of type **DOUBLE PRECISION**, and sets `num` equal to `3.141592654`:

```
DOUBLE PRECISION pi / 3.141592654 /
```

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.

A **DOUBLE PRESICION** variable is accurate to 14 or 15 decimal digits.

DOUBLE PRECISION and **REAL*8** are the same data type.

EXAMPLE

```
DOUBLE PRECISION  varnam
```

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 **END IF** statement at the same **IF** level. The matching **END IF** statement must appear before any **ELSE** or **ELSE IF** statements of the same **IF** level.

Control may not be transferred into an **ELSE** block from outside that block.

EXAMPLE

```
CHARACTER c
.
.
.
READ (*, '(A)') C
IF (c' .EQ. 'A') THEN
  CALL Asub
ELSE
  CALL Other
END IF
.
.
.
```

SEE ALSO **ELSE IF**
END IF
IF THEN ELSE (Block **IF**)

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

SYNTAX **ELSE IF** (*expression*) **THEN**

Parameter	Description
expression	A logical expression

REMARKS The associated **ELSE IF** block consists of any executable statements between the **ELSE IF** statement and the next **ELSE IF**, **ELSE**, or **END IF** statement at the same **IF** level.

When the **ELSE IF** statement is executed, *expression* is evaluated, and the following steps are performed:

If *expression* is:	Then the next statement executed is:
True, and there is at least one executable statement in the **ELSE IF** block	The first statement of the **ELSE IF** block
True, and there are no executable statements in the **ELSE IF** block	The next **END IF** statement at the same **IF** level as the **ELSE IF** statement
False	The next **ELSE IF**, **ELSE**, or **END IF** statement that has the same **IF** level as the **ELSE IF** statement

After the last statement in the **ELSE IF** block has been executed, the next statement executed is the next **END IF** statement at the same **IF** level as that **ELSE IF** statement.

Control may not be transferred into an **ELSE IF** block from outside that block.

EXAMPLE

```
CHARACTER char
READ (*, '(A)') char
IF (char .EQ. 'L') THEN
  CALL Lsub
ELSE IF (char .EQ. 'X') THEN
  CALL Xsub
ELSE
  CALL Other
END IF
```

SEE ALSO

ELSE
END IF
IF THEN ELSE (Block **IF**)

ACTION Terminates execution of the main program, or returns control from a subprogram

SYNTAX **END**

REMARKS The **END** statement marks the end of the program unit in which it appears, and it must be the last statement in every program unit. Comment lines may follow an **END** statement.

An **END** statement must appear by itself on an unlabeled initial line (not a continuation line). No continuation lines may follow an **END** statement. No FORTRAN statement may have an initial line that appears to be an **END** statement (such as splitting an **END IF** statement over two lines).

In a subprogram, the **END** statement has the same effect as a **RETURN** statement.

EXAMPLE

```
C     An END statement must be the last statement in a program
C     unit:
      PROGRAM MyProg
      WRITE (*, '("Hello, world!")')
      END
```

ACTION Terminates a **DO** or **DO WHILE** loop

SYNTAX **END DO**

REMARKS There must be a matching **END DO** statement for every **DO** or **DO WHILE** statement that does not contain a label reference.

An **END DO** statement may terminate only one **DO** or **DO WHILE** statement.

EXAMPLES

The following examples produce the same output.

```
DO ivar = 1, 10
   PRINT ivar
END DO

ivar = 0
DO WHILE (ivar .LT. 10)
   ivar = ivar + 1
   PRINT ivar
END DO
```

ACTION Terminates a block **IF** statement

SYNTAX **END IF**

REMARKS There must be a matching **END IF** statement for every block **IF** statement in a program unit. Execution of an **END IF** statement has no effect on the program.

EXAMPLE

```
IF (n .LT. 0) THEN
  x = -n
  y = -n
END IF
```

ACTION

Writes an end-of-file record to the specified unit

SYNTAX

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

If **UNIT=** is omitted, *unitspec* must be the first parameter. The parameters can otherwise appear in any order.

Parameter	Description
unitspec	An integer expression that specifies an external unit. If *unitspec* has not been opened, a run-time error occurs.
errlabel	The label of an executable statement in the same program unit. If *errlabel* is specified, an I/O error causes transfer of control to the statement at *errlabel*. If *errlabel* is omitted, the effect of an I/O error is determined by the presence or absence of *iocheck*.
iocheck	An integer variable, array element, or structure element that returns the value zero if there is no error, or the error number if an error occurs. For more information on error handling, see Section 3.2.6.

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.

When **ENDFILE** operates on a direct-access file, all records beyond the new end-of-file record are erased.

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 unpredictable results can occur.

EXAMPLE

```
WRITE (6, *) x
ENDFILE 6
REWIND 6
READ (6, *) y
```

ACTION Specifies an entry point to a subroutine or external function

SYNTAX **ENTRY** *ename* [[[*eattrs*]]] [[([[*formal* [[[*attrs*]]] [[, *formal* [[[*attrs*]]]]]...])]]

Parameter	Description
ename	The name of the entry point. If *ename* is an entry point for a user-defined function, *ename* must be given a data type by: ■ The default rules used in determining type; or ■ The type specified in an **IMPLICIT** statement; or ■ A declaration in the user-defined function's type-declaration section.
eattrs	A list of attributes, separated by commas. The *eattrs* describe *ename*. The following attributes can be used with *ename*: **ALIAS, C, LOADDS, PASCAL, VARYING.**
formal	A variable name, array name, structure variable 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 an **ENTRY** statement, replace the name of the subprogram with the name of the entry point:

Type of Call	Form of Call
Subroutine	**CALL** *ename* [[([[*actual1* [[, *actual2*]] ...]])]]
Function	ename ([[*actual1* [[, *actual2*]] ...]])

Parentheses are required when calling a function entry point, even if the function has no formal arguments.

Entry points cannot be called recursively. That is, a subprogram may not directly or indirectly call an entry point within that subprogram.

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 have the same name as a *formal* argument in a **FUNCTION, SUBROUTINE, ENTRY,** or **EXTERNAL** statement.
- Within a function, *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 is of character type, all the *enames* in that function that must be of character type, and all the *enames* must be the same length.
- A formal argument cannot appear in an executable statement that occurs before the **ENTRY** statement containing the formal argument unless the formal argument also appears in a **FUNCTION, SUBROUTINE,** or **ENTRY** statement that precedes the executable statement.
- An **ENTRY** statement cannot appear between a block **IF** statement and the corresponding **END IF** statement, or between a **DO** statement and the terminal statement of its **DO** loop.

EXAMPLE

```
C     This fragment writes a message indicating
C     whether num is positive or negative
      IF (num .GE. 0) THEN
          CALL Positive
      ELSE
          CALL Negative
      END IF
      .
      .
      .
      END

      SUBROUTINE Sign
      ENTRY Positive
      WRITE (*, *) 'It''s positive.'
      RETURN
      ENTRY Negative
      WRITE (*, *) 'It''s negative.'
      RETURN
      END
```

ACTION Causes two or more variables or arrays to occupy 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. The list may not include formal arguments or allocatable arrays. Subscripts must be integer constants and must be within the bounds of the array they index. An unsubscripted array name refers to the first element of the array.

REMARKS An **EQUIVALENCE** statement causes all elements in *nlist* to have the same first memory location. Variable names are said to be associated if they refer to the same memory location.

There is no automatic type conversion among the elements in *nlist*; they simply occupy the same memory space.

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 memory location, nor can you force two or more elements of the same array to occupy the same memory location. The following statement would force `r` to occupy two different 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))
      EQUIVALENCE (r, s(2))
```

- Consecutive array elements must be stored in sequential order. The following is not permitted:

```
C     this causes another error:
      REAL r(10), s(10)
      EQUIVALENCE (r(1), s(1))
      EQUIVALENCE (r(5), s(7))
```

  The compiler always aligns noncharacter entities on even-byte (word) boundaries. The following example causes a compile-time error, since variables `a` and `b` cannot be word-aligned simultaneously:

```
      CHARACTER*1 c1(10)
      REAL a, b
      EQUIVALENCE (a, c1(1))
      EQUIVALENCE (b, c1(2))
```

- Character and noncharacter entities cannot be associated. Microsoft FORTRAN permits character and noncharacter entities to be associated, but not in such a way that noncharacter entities start on an odd-byte boundary. If necessary, the compiler will adjust the storage location of the character entity so the noncharacter entity begins on an even byte. The following example causes a compile-time error because it is not possible to reposition the character array such that both noncharacter entities start on an even byte address:

```
      CHARACTER 1  char1(10)
      REAL  reala, realb
      EQUIVALENCE  (reala, char1(1))
      EQUIVALENCE  (realb, char1(2))
```

- 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 share memory between two different common blocks or between elements of the same common block.

- 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(7))
```

Unless the **$STRICT** metacommand is in effect, only the first subscript of a multi-dimensional array is required in **EQUIVALENCE** statements. This makes it easier to port FORTRAN 66 programs.

For example, the array declaration `var(3,3), var(4)` could appear in an **EQUIVALENCE** statement. The reference is to the fourth element of the array (`var(1,2)`), not to the beginning of the fourth row or column.

EXAMPLE

```
CHARACTER  name, first, middle, last
DIMENSION  name(60), first(20), middle(20), last(20)
EQUIVALENCE (name(1), first(1)), (name(21), middle(1))
EQUIVALENCE (name(41), last(1))
```

ACTION

Transfers control from within a **DO** or **DO WHILE** loop to the first executable statement following the end of the loop

SYNTAX

EXIT

REMARKS

Normally, a **DO** loop executes a fixed number of times. The **EXIT** statement lets you terminate the loop early if some specified condition warrants it.

For example, you could set a loop to collect a maximum of 1,000 data points, while still being able to terminate the loop if there were fewer than 1,000 points.

EXAMPLE

```
C     Loop terminates early if one of the data points is zero:

      INTEGER numpoints, point
      REAL datarray(1000), sum

      sum = 0.0
      DO point = 1, 1000
         numpoints = point
         sum = sum + datarray(point)
         IF (datarray(point+1) .EQ. 0.0)  EXIT
      END DO
```

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

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

Parameter	Description
name	The name of an external subroutine or function. Statement-function (single-line function) names are not allowed.
attrs	A list of attributes, separated by commas. The *attrs* describe *name*. The following attributes can be used with *name*: **ALIAS, C, FAR, LOADDS, NEAR, PASCAL, VARYING.**

REMARKS The **EXTERNAL** statement is primarily used to specify that a particular user-defined name is a subroutine or function to be used as a procedural parameter. The **EXTERNAL** statement can also replace an intrinsic function with a user-defined function of the same name.

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.

FORTRAN assumes that the name of any subroutine or function referred to in a compilation unit (but not defined there) is defined externally.

EXAMPLES

```
      EXTERNAL MyFunc, MySub
C     MyFunc and MySub are arguments to Calc
      CALL Calc (MyFunc, MySub)

C     Example of a user-defined function replacing an
C     intrinsic
      EXTERNAL SIN
      x = SIN (a, 4.2, 37)
```

ACTION Sets the format in which data is written to or read from a file

SYNTAX **FORMAT** ([[*editlist*]])

Parameter	Description
editlist	A list of editing descriptions

REMARKS **FORMAT** statements must be labeled.

Invalid *editlist* strings generate error messages; some during compilation, others at run-time.

Table 3.6 summarizes the nonrepeatable edit descriptors. Table 4.5 summarizes the repeatable edit descriptors. See Section 3.7, "Formatted I/O," for further information on edit descriptors and formatted I/O.

Table 4.5 Repeatable Edit Descriptors

Descriptor[a]	Use
I*w*[[.*m*]]	Integer values
Z*w*	Hexadecimal values
F*w.d*	Real values
E*w.d*[[E*e*]]	Real values with exponents
G*w.d*[[E*e*]]	Real values, extended range
D*w.d*	Double-precision real values
L*w*	Logical values
A[[*w*]]	Character values

[a] In this column *w* represents width of field, *m* represents number of leading zeroes, *d* represents number of digits following the decimal point, and *e* represents the exponent.

ACTION Identifies a program unit as a function, and supplies its name, data type, and optional formal parameters

SYNTAX [[*type*]] **FUNCTION** *func* [[(*fattrs*)]] ([[*formal* [[(*attrs*)]]]][[,*formal*[[(*attrs*)]]]]...)

Parameter	Description
type	Declares the data type of the value returned by the function. The *type* parameter must be one of the following: **CHARACTER** **CHARACTER*n** **COMPLEX** **COMPLEX*8** **COMPLEX*16** **DOUBLE COMPLEX** **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 the applicable **IMPLICIT** statements. If there are none, the function's type is established by FORTRAN's default typing. If **IMPLICIT NONE** or the **$DECLARE** metacommand is in effect, the function's type must be given in the function declaration or in a type declaration. If *type* is specified, the function name cannot appear in a type statement.
func	The name of the function. The name *func* cannot appear in **AUTOMATIC**, **COMMON**, **DATA**, **EQUIVALENCE**, **INTRINSIC**, **NAMELIST**, or **SAVE** statements.
fattrs	A list of attributes, separated by commas. The *fattrs* describe *func*. The following attributes can be used with the *func* name: **ALIAS**, **C**, **FAR**, **NEAR**, **PASCAL**, **VARYING**.

formal — A formal-argument name. If more than one is specified, they must be separated by commas. Alternate-return specifiers are not allowed in **FUNCTION** 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

A function begins with a **FUNCTION** statement and ends with the next **END** statement. A function can contain any statement except a **BLOCK DATA, FUNCTION, INTERFACE TO, PROGRAM,** or **SUBROUTINE** statement.

The length specifier for the function type may follow the function name. For example, the following two declarations are equivalent:

```
INTEGER*4  FUNCTION  FuncX   (var)
INTEGER    FUNCTION  FuncX*4 (var)
```

Within the calling program, *func* is global, and may not be used for any other variable or subprogram.

The formal-argument list sets the number of arguments for that function. The argument types are set by any **IMPLICIT, EXTERNAL, DIMENSION,** or type statements within the function itself (or they default to implicit FORTRAN types). The argument types are not set by the calling program, even if the function appears in the calling program's source file. Formal-argument names cannot appear in **AUTOMATIC, COMMON, DATA, EQUIVALENCE, INTRINSIC,** or **SAVE** statements.

When a function is referenced, the actual arguments that are passed must agree with corresponding formal arguments in the **FUNCTION** 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 the **CALL** entry in this section for more information on checking arguments for correspondence.

The function name *func* acts as if it were a variable within the function definition. At some point in the execution of a function, a value must be assigned to the function name. It is this value that is returned to the calling program through the function's name. This value is usually assigned at the end of the function (but any number of assignments may be made), with **RETURN** statements terminating the function at the appropriate points.

In addition to returning the value of the function, an external function can return values by assignment to any formal argument whose corresponding actual argument was passed by reference.

A "recursive" function is one that calls itself, or calls another subprogram, which in turn calls the first function before the first function has completed execution. FORTRAN does not support recursive function calls. A function may otherwise be called from any unit.

EXAMPLE

```
C     GetNo is a function that reads a number from unit i
      i = 2
      10 IF (GetNo(i) .EQ. 0.0)  GOTO 10
      END
C
      FUNCTION GetNo (nounit)
      READ (nounit, '(F10.5)') r
      GetNo = r
      END
```

ACTION

Transfers execution to the statement label assigned to *variable*

SYNTAX

GOTO *variable* [[[,]] (*labels*)]]

Parameter	Description
variable	An integer variable. The *variable* must have been assigned the label of an executable statement in the same program unit.
labels	A list of one or more statement labels of executable statements in the same program unit. If more than one label is specified, the labels are separated by commas. A *label* may appear more than once in a list.

REMARKS

If you specify the **$DEBUG** metacommand, a run-time error occurs if the label assigned to *name* is not one of the labels specified in the *label* list.

Jumping into a **DO**, **IF**, **ELSE IF**, or **ELSE** block from outside the block is not normally permitted. A special feature, extended-range **DO** loops, does permit entry into a **DO** block. See the **$DO66** entry in Section 6.2, "Metacommand Directory," for more information.

EXAMPLES

```
      ASSIGN 10 TO n
      GOTO n
   10 CONTINUE
```

The following example uses an assigned **GOTO** statement to check the value of `clearance`:

```
$DEBUG

      INTEGER  view, clearance
C     Assign an appropriate label to view:
      IF (clearance .EQ. 1) THEN
         ASSIGN 200 TO view
      ELSE IF (clearance .EQ. 2) THEN
         ASSIGN 400 TO view
      ELSE
         ASSIGN 100 TO view
      END IF

C     Show user appropriate view of data depending on
C     security clearance.
      GOTO view (100, 200, 400)
```

```
C     If view had not been assigned one of the valid labels
C     (100, 200, or 400), a run-time error would have occurred.

 100  CONTINUE
      .
      .
      .
 200  CONTINUE
      .
      .
      .
 400  CONTINUE
      .
      .
      .
      END
```

ACTION Transfers control to the statement at the *n*th label in the list

SYNTAX **GOTO** (*labels*) [[,]] *n*

Parameter	Description
labels	One or more statement labels of executable statements in the same program unit. If there is more than one label, the labels are separated by commas. The same statement label may appear more than once in the list.
n	An integer expression. Control is transferred to the *n*th label in the list.

REMARKS If there are *j* labels in the list and *n* is out of range (that is, $n > j$ or $n < 1$), the computed **GOTO** statement acts like a **CONTINUE** statement.

Jumping into a **DO**, **IF**, **ELSE IF**, or **ELSE** block from outside the block is not normally permitted. A special feature, extended-range **DO** loops, does permit entry into a **DO** block. See the **$DO66** entry in Section 6.2,"Metacommand Directory," for more information.

EXAMPLE

```
      next = 1
C     The following statement transfers control to statement 10:
      GOTO (10, 20) next
      .
      .
      .
   10 CONTINUE
      .
      .
      .
   20 CONTINUE
```

ACTION Transfers control to a labeled statement

SYNTAX **GOTO** *label*

Parameter	Description
label	The statement label of an executable statement in the same program unit

REMARKS If the program unit does not contain a statement with the label specified in the GOTO statement, a compile-time error occurs.

Jumping into a **DO**, **IF**, **ELSE IF**, or **ELSE** block from outside the block is not normally permitted. A special feature, extended-range **DO** loops, does permit entry into a **DO** block. See the **$DO66** entry in Section 6.2, "Metacommand Directory," for more information.

EXAMPLE

```
      GOTO 4077
      .
      .
      .
4077  CONTINUE
```

ACTION

Transfers control to one of three statement labels, depending on the value 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*	The statement labels of executable statements in the same program unit. The same statement label may appear more than once.

REMARKS

The arithmetic **IF** statement transfers control as follows:

If *expression* is:	The next statement executed is:
< 0	The statement at *label1*.
= 0	The statement at *label2*.
> 0	The statement at *label3*.

Jumping into a **DO**, **IF**, **ELSE IF**, or **ELSE** block from outside the block is not normally permitted. A special feature, extended-range **DO** loops, does permit entry into a **DO** block. See the **$DO66** entry in Section 6.2,"Metacommand Directory," for more information.

EXAMPLE

```
      DO 40 j = -1, 1
      n = j
C     The following statement transfers control to statement 10
C     for j = -1, to statement 20 for j = 0, and to statement 30
C     for j = +1.
      IF (n) 10, 20, 30
   10 CONTINUE
      .
      .
      .
   20 CONTINUE
      .
      .
      .
   30 CONTINUE
      .
      .
      .
   40 CONTINUE
```

ACTION

If *expression* is true, *statement* is executed; if *expression* is false, program execution continues with the next executable statement

SYNTAX

IF (*expression*) *statement*

Parameter	Description
expression	A logical expression.
statement	Any executable statement except a **CASE**, **DO**, **ELSE**, **ELSE IF**, **END**, **END IF**, **END SELECT CASE**, **SELECT CASE**, block **IF**, or another logical **IF** statement. Note that the statement can be an arithmetic **IF**.

EXAMPLES

```
      IF (i .EQ. 0)  j = 2

      IF (x .GT. 2.3)  GOTO 100
      .
      .
      .
100   CONTINUE
```

ACTION

If *expression* is true, statements in the **IF** block are executed; if *expression* is false, control is transferred to the next **ELSE**, **ELSE IF**, or **END IF** statement at the same **IF** level

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 **ELSE IF**, **ELSE**, or **END IF** statement at the same **IF** level as this block **IF** statement. (**IF** levels are defined below.)

The block **IF** statement transfers control as follows:

If *expression* is:	Then the next statement executed is:
True, and the **IF** block has no executable statements	The next **END IF** statement at the same **IF** level as the block **IF** statement
True, and there is at least one executable statement in the **IF** block	The first executable statement in the **IF** block
False	The next **ELSE IF**, **ELSE**, or **END IF** 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 **END IF** 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.

The **IF** level of a program statement is *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 **END IF** statements from the beginning of the program unit up to, but not including, that statement.

Item	Required IF-Level Value
Block **IF**, **ELSE IF**, **ELSE**, and **END IF**	Greater than 0 and less than approximately 50
END	0
Other statements	Greater than or equal to 0 and less than approximately 50

EXAMPLES

```
C     Simple block IF that skips a group of statements
C     if the expression is false:
      IF (i .LT. 10) THEN
C     the next two statements are only executed if i is < 10
         j = i
         slice = TAN (angle)
      END IF

C     Block IF with ELSE IF statements:

      IF (j .GT. 1000) THEN
C     Statements here are executed only if J > 1000
      ELSE IF (j .GT. 100) THEN
C     Statements here are executed only if J > 100 and j <= 1000
      ELSE IF (j .GT. 10) THEN
C     Statements here are executed only if J > 10 and j <= 100
      ELSE
C     Statements here are executed only if j <= 10
      END IF

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

      IF (i .LT. 100) THEN
C     Statements here are executed only if i < 100
         IF (j .LT. 10) THEN
C     Statements here are executed only if i < 100 and j < 10
         END IF
C     Statements here are executed only if i < 100
      ELSE
C     Statements here are executed only if i >= 100
         IF (j .LT. 10) THEN
C     Statements here are\executed only if i >= 100 and j < 10
         END IF
C     Statements here are executed only if i >= 100
      END IF
```

SEE ALSO

ELSE
ELSE IF
END IF

ACTION Defines the default type for user-declared names

SYNTAX **IMPLICIT** { *type* (*letters*) [[, *type* (*letters*)]]... | **NONE** }

Parameter	Description
type	One of the following types: **CHARACTER** **CHARACTER*n** **COMPLEX** **COMPLEX*8** **COMPLEX*16** **DOUBLE COMPLEX** **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. Case is not significant. 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. In this context, Microsoft FORTRAN allows the use of the dollar sign ($) as an alphabetic character that follows the letter Z.

REMARKS An **IMPLICIT** statement establishes the default type for all user-defined names that begin with any of the specified letters. An **IMPLICIT** statement applies only to the program unit in which it appears and does not change the type of any intrinsic function.

If **IMPLICIT NONE** is specified, then all user-defined names must be explicitly typed. An untyped name causes a compile-time error.

An explicit type statement can confirm or override the type established by an **IMPLICIT** statement. An explicit type in a **FUNCTION** statement also overrides the type indicated by an **IMPLICIT** statement. If the type in question is a character type, the length may be 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. A particular letter cannot appear in more than one **IMPLICIT** statement in the same program unit.

A program unit can have only one **IMPLICIT NONE** statement. No other **IMPLICIT** statements may appear in a program unit that contains an **IMPLICIT NONE** statement.

EXAMPLE

```
      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'
```

ACTION Inserts the contents of a specified text file at the location of the **INCLUDE** statement

SYNTAX **INCLUDE** *'filename'*

Parameter	Description
filename	The name of the FORTRAN text file to include in the program, surrounded by apostrophes.

REMARKS The argument *filename* must be a valid text file specification for your operating system.

The compiler considers the contents of the include file to be part of the program file and compiles them immediately. At the end of the include file, the compiler resumes processing the original source file at the line following the **INCLUDE** statement.

Include files are primarily used for data or program units that appear in more than one program. Include files most often contain subroutines and functions, common block declarations, and **EXTERNAL**, **INTERFACE**, and **INTRINSIC** statements.

Include files can also contain other **INCLUDE** statements and **$INCLUDE** metacommands; this is called "nesting" included files. The compiler allows you to nest any combination of up to ten **INCLUDE** statements or **$INCLUDE** metacommands. Your operating system may impose further restrictions.

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 in the following program) are inserted in the source code in place of every **INCLUDE** statement. 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 out the numbers in reverse order:
      WRITE (*, *) ' '
      DO 200 i = 1, 5
         CALL Pop (x)
         WRITE (*, *) x
200   CONTINUE
      END
```

```
      SUBROUTINE Push (x)
C     Pushes an element x onto the top of the stack.

      REAL x
      INCLUDE 'stkvars.for'

      top = top + 1
      IF (top .GT. stacksize)  STOP 'Stack overflow'
      stack(top) = x
      END

      SUBROUTINE pop(x)
C     Pops an element from the top of the stack into x.

      REAL x
      INCLUDE 'stkvars.for'

      IF (top .LE. 0)  STOP 'Stack underflow'
         x   = stack(top)
         top = top - 1
      END
```

The following is the text file STKVARS.FOR:

```
C     This file contains the declaration of the common block
C     for a stack implementation. Because this file contains an
C     assignment statement, it must be included only after all
C     other specification statements in each program unit.

      REAL     stack(500)
      INTEGER  top, stacksize

      COMMON  /stackbl/ stack, top

      stacksize = 500
```

ACTION Returns the properties of a unit or external 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*]]
[[, **IOFOCUS=***iofocus*]]
[[, **IOSTAT=***iocheck*]]
[[, **MODE=***mode*]]
[[, **NAME=***name*]]
[[, **NAMED=***named*]]
[[, **NEXTREC=***nextrec*]]
[[, **NUMBER=***num*]]
[[, **OPENED=***opened*]]
[[, **RECL=***recl*]]
[[, **SEQUENTIAL=***seq*]]
[[, **SHARE=***share*]]
[[, **UNFORMATTED=***unformatted*]])

If **UNIT=** is omitted, *unitspec* must be the first parameter. The parameters can otherwise appear in any order.

Parameter	Description
unitspec	The *unitspec* parameter can be either an integer expression or an asterisk (*). If **UNIT** = * is specified, you may not include the **NUMBER=** option, or a run-time error occurs. Exactly one *unitspec* or *file* must be specified, but not both. If *unitspec* is given, the inquiry is an "inquire-by-unit" operation.
file	A character expression that the operating system recognizes as the name of the file you are interested in. Exactly one *unitspec* or *file* must be specified, but not both. If *file* is given, the inquiry is an "inquire-by-file" operation.

access — A character variable, array element, or structure element. Returns **'SEQUENTIAL'** if the specified unit or file is open for appending data. Returns **'SEQUENTIAL'** if the specified unit or file is connected for sequential access. Returns **'DIRECT'** if the specified unit 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, array element, or structure element. Returns **'YES'** if binary is among the allowable forms for the specified file or unit. Returns **'NO'** or **'UNKNOWN'** otherwise.

blank — A character variable, array element, or structure element. Returns **'NULL'** if the **BN** edit descriptor is in effect, and returns **'ZERO'** if **BZ** is in effect.

blocksize — An integer variable, array element, or structure element. If the *unitspec* or *file* is connected, *blocksize* returns the I/O buffer size. If *unitspec* or *file* is not connected, *blocksize* is undefined.

In a QuickWin application, *blocksize* returns the size of the screen buffer.

direct — A character variable, array element, or structure element. Returns **'YES'** if *direct* is among the allowable access modes for the specified file or unit. Returns **'NO'** or **'UNKNOWN'** otherwise.

errlabel — The label of an executable statement in the same program unit. If *errlabel* is specified, an I/O error causes transfer of control to the statement at *errlabel*. If *errlabel* is omitted, the effect of an I/O error is determined by the presence or absence of *iocheck*.

exist — A logical variable, array element, or structure element. Returns **.TRUE.** if the specified unit or file exists; returns **.FALSE.** otherwise.

form — A character variable or array element. Returns **'FORMATTED'** if the specified unit or file is connected for formatted I/O; returns **'UNFORMATTED'** for unformatted I/O; returns **'BINARY'** for binary I/O.

formatted — A character variable, array element, or structure element. Returns **'YES'** if formatted is among the allowable forms for the specified file or unit; returns **'NO'** or **'UNKNOWN'** otherwise.

iocheck — An integer variable, array element, or structure element that returns a value of zero if there is no error, or the number of the error message if an error occurs. For more information on error handling, see Section 3.2.6, "Error and End-of-File Handling."

iofocus — A logical variable, array element, or structure element that returns a value of **.TRUE.** if the specified unit has the current I/O focus (is the active window) in a QuickWin application. If the unit does not have the current I/O focus, the return value is **.FALSE.**

mode — A character variable, array element, or structure element that returns the current mode status of the specified file or unit. The returned modes 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, array element, or structure element. In an inquire-by-unit operation, returns 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, returns the name specified for *file*.

named — A logical variable, array element, or structure element. Returns **.FALSE.** if the file specified by *file* or attached to *unitspec* is not open or if it is a scratch file; returns **.TRUE.** otherwise.

nextrec — An integer variable, array element, or structure element that returns 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, array element, or structure element. In an inquire-by-file operation, returns the number of the unit connected to *file*. If no unit is connected to *file*, *num* is undefined. In an inquire-by-unit operation, returns the number given in *unitspec*. If you specify **UNIT** = *, do not include the option **NUMBER=**, or a run-time error occurs.

opened — A logical variable, array element, or structure element. In an inquire-by-unit operation, returns **.TRUE.** if any file is currently connected to *unitspec*; returns **.FALSE.** otherwise. In an inquire-by-file operation, returns **.TRUE.** if *file* is currently connected to any unit; returns **.FALSE.** otherwise.

recl	An integer variable, element name, or structure element that returns the length (in bytes) of each record in a direct-access file. If the file is connected for unformatted I/O, the value is in processor-dependent units. (In Microsoft FORTRAN, this is also the number of bytes.)
seq	A character variable, array element, or structure element. Returns **'YES'** if sequential is among the allowable access modes for the specified file or unit; returns **'NO'** or **'UNKNOWN'** otherwise.
share	A character variable, array element, or structure element. Returns the current share status of the specified file or unit. The returned values 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, array element, or structure element. Returns **'YES'** if unformatted is among the allowable forms for the specified file or unit; returns **'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 return the properties of an unopened file, and it cannot distinguish between attributes that are specified by you and attributes that are set by default.

You can execute the **INQUIRE** statement at any time. It returns the values 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 execute an I/O statement or the **EOF** intrinsic function because the results are unpredictable.

EXAMPLE

```
C     This program prompts for the name of a data file. The INQUIRE
C     statement then determines whether or not the file exists.
C     If it does not, the program prompts for another file name.

      CHARACTER*12  fname
      LOGICAL       exists

C     Get the name of a file:
  100 WRITE (*, '(1X, A\)') 'Enter the file name: '
      READ  (*, '(A)') fname

C     INQUIRE about file's existence:
      INQUIRE (FILE = fname, EXIST = exists)

      IF (.NOT. exists) THEN
         WRITE (*,'(2A/)') ' > Cannot find file ', fname
         GOTO 100
      END IF
      .
      .
      .
      END
```

ACTION Specifies the **INTEGER** type for 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. Specifies the length, in bytes, of the names in 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, ALLOCATABLE, C, EXTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE.**
length	Must be 1, 2, or 4. Gives *vname* the specified *length*. The *length* parameter overrides the length specified by *bytes*.
dim	A dimension declarator. Specifying *dim* declares *vname* as an array.
values	A list of constants and repeated constants, separated by commas. A repeated constant takes the form *n*constant*, where *n* is a positive integer constant. The */values/* option initializes *vname*. The following statement 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.

If both the *bytes* and *length* parameters are omitted, the variable's length defaults to the value given in the **$STORAGE** metacommand.

INTEGER statements must precede all executable statements.

EXAMPLES

```
INTEGER count, matrix(4, 4), sum*2
INTEGER*2 q, m12*2, ivec*2(10), z*24(10)
```

ACTION Specifies the name of a subroutine or function, its attributes and formal argument types

SYNTAX

INTERFACE TO {*function-statement* | *subroutine-statement*}
(*Formal-parameter-declaration(s)*)
END

Parameter	Description
function-statement	A function declaration statement
subroutine-statement	A subroutine declaration statement

REMARKS

An interface has two parts. The first is the **INTERFACE TO** statement followed by a **FUNCTION** or **SUBROUTINE** declaration statement. The second is a list of declarations for all the procedure's formal arguments. Only type, **DIMENSION, EXTERNAL, INTRINSIC,** and **END** statements can appear in the second part.

The **INTERFACE TO** statement confirms the correctness of subprogram calls. The compiler verifies that the number and types of arguments in a subprogram call are consistent with those in the interface. (Note that the interface must appear prior to any subprogram references for the checking to work.)

The attributes 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 also appear in the **INTERFACE TO** statement.

EXAMPLE

```
INTERFACE TO INTEGER FUNCTION Func (p, d, q)
INTEGER*2 p
REAL d[C]          ! required; appears in function definition
REAL*8 q[FAR]      ! overrides default in function definition
END
```

The interface above describes the following function to the calling unit:

```
INTEGER FUNCTION Func (r, s, t)
INTEGER*2 r
REAL s[C]
REAL*8 t
.
.
.
END
```

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

A name may appear only once in an **INTRINSIC** statement. A name appearing in an **INTRINSIC** statement cannot also 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 Chapter 5, "Intrinsic Functions and Additional Procedures."

EXAMPLE

```
      INTRINSIC SIN, COS
C     SIN and COS are arguments to Calc2:
      result = Calc2 (SIN, COS)
```

ACTION Locks 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**=*rec*]]
[[, **RECORDS**=*recnum*]])

If **UNIT**= is omitted, *unitspec* must be the first parameter. The parameters can otherwise appear in any order.

Parameter	Description
unitspec	An integer expression for the unit to be locked. The file attached to *unitspec* must have been opened for direct access.
errlabel	The label of an executable statement in the same program unit. If *errlabel* is specified, an I/O error causes transfer of control to the statement at *errlabel*. If *errlabel* is omitted, the effect of an I/O error is determined by the presence or absence of *iocheck*.
iocheck	An integer variable, array element, or structure element that returns the value zero if there is no error, or the number of the error message if an error occurs. For more information on error handling, see Section 3.2.6, "Error and End-of-File Handling."
lockmode	A character expression with one of the following values:

Value	Description
'NBLCK'	Locks the specified region for reading and writing. If any of the records are already locked by a different process, a run-time error occurs. This non-blocking lock is the default.
'LOCK'	Locks the specified region for reading and writing. Waits for any part of the region locked by a different process to become available.
'NBRLCK'	Locks the specified region for reading; records may still be written. This non-blocking read lock is otherwise the same as **'NBLCK'**.

	'RLCK'	Locks the specified region for reading; records may still be written. This read lock is otherwise the same as **'LOCK'**.
	'UNLCK'	Unlocks the specified region.
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 one.

REMARKS

The **LOCKING** statement has no effect when used with versions of MS-DOS before 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.

ACTION Specifies the **LOGICAL** type for 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. Specifies the length, in bytes, of the names in 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, ALLOCATABLE, C, EXTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE.**
length	Must be 1, 2, or 4. Gives *vname* the specified *length*. The *length* parameter overrides the length specified by *bytes*.
dim	A dimension declarator. Specifying *dim* declares *vname* as an array.
values	A list of constants and repeated constants, separated by commas. A repeated constant takes the form *n*constant*, where *n* is a positive-nonzero-integer constant, and is equivalent to *constant* repeated *n* times. The */values/* option initializes *vname*. The following statement 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.

If both the *bytes* and *length* parameters are omitted, the variable's length defaults to the value given in the **$STORAGE** metacommand.

LOGICAL statements must precede all executable statements.

EXAMPLE

```
LOGICAL  switch
LOGICAL*1  flag
```

ACTION Delimits a group of variable type declarations within a **STRUCTURE** statement

SYNTAX

MAP
type-declaration(s)
.
.
.
END MAP

Parameter	Description
type-declaration	A variable declaration; can be any conventional type (**INTEGER**, **REAL*8**, **CHARACTER**, etc.), or a **RECORD** declaration of a structure type

REMARKS A **MAP...END MAP** block must appear within a **UNION...END UNION** block.

Any number of variables may appear within a map, and they may be a mixture of types. Variables are stored in memory contiguously, in the same order they appear in the **MAP** statement.

There are always at least two maps within a union. The purpose of a union is to assign two or more maps to the same memory location. Unlike the FORTRAN **EQUIVALENCE** statement which overlays single variables or arrays, the **MAP** statement allows groups of variables to overlay other groups in the same union.

For example, the two maps in the following union each contain three **REAL*8** variables. The variable `manny` shares the same memory space as `meg`, `moe` shares with `jo`, and `jack` shares with `amy`.

```
UNION
   MAP
      REAL*8  meg, jo, amy
   END MAP
   MAP
      REAL*8  manny, moe, jack
   END MAP
END UNION
```

EXAMPLES

Within a structure, maps can contain variables of intrinsic type:

```
UNION
   MAP
      CHARACTER*20 string
   END MAP
   MAP
INTEGER*2 number(10)
   END MAP
END UNION
```

Or they can contain variables with types defined in previous **STRUCTURE** statements:

```
UNION
   MAP
      RECORD /Cartesian/ xcoord, ycoord
   END MAP
   MAP
      RECORD /Polar/ length, angle
   END MAP
END UNION
```

SEE ALSO

STRUCTURE...END STRUCTURE
UNION...END UNION

ACTION

Declares a group name for a set of variables so they may be read or written with a single namelist-directed **READ** or **WRITE** statement

SYNTAX

NAMELIST */namlst/ varlist* [[*/namlst/ varlist*]]

Parameter	Description
namlst	The name for a group of variables
varlist	A list of variable names and array names; may not include structure variables and formal arguments

REMARKS

The same variable name may appear in more than one namelist.

Any namelist can appear more than once in the same program unit. The list of elements in a particular namelist is treated as a continuation of the list in the previous namelist with the same name.

If the namelist is used in a **WRITE** statement, the list of variable names followed by their values is written to a file or displayed.

If the namelist is used in a **READ** statement, the file connected to *unit* or the user's input is scanned for the names in the namelist and their corresponding values. The order in which the names appear is not important, nor do all the names in the namelist have to appear.

Although a namelist is limited to single variables and complete arrays, a file to be read may include the names of individual array elements and character variable substrings.

Note that when you use a namelist in a **READ** statement, the program expects the variable names you enter to exactly match the names you gave in the **NAMELIST** statement. However, when the **$TRUNCATE** metacommand is used, the compiler truncates all variable names to six characters. In this circumstance, the program may not recognize the original, untruncated name; you must enter the shortened version of the name.

For example, if one of your **NAMELIST** variables is `impedance` and you use the **$TRUNCATE** metacommand, a **READ** statement using the namelist expects `impeda`, not `impedance`.

EXAMPLE

```
      INTEGER    i1*1, i2*2, i4*4, iarray(3)
      LOGICAL    l1*1, l2*2, l4*4
      REAL       r4*4, r8*8
      COMPLEX    z8*8, z16*16
      CHARACTER  c1*1, c10*10

      NAMELIST /example/ i1, i2, i4, l1, l2, l4, r4, r8,
     +                   z8, z16, c1, c10, iarray

      i1  =  11
      i2  =  12
      i4  =  14
      l1  = .TRUE.
      l2  = .TRUE.
      l4  = .TRUE.
      r4  =  24.0
      r8  =  28.0d0
      z8  = (38.0, 0.0)
      z16 = (316.0d0, 0.0d0)
      c1  = 'A'
      c10 = '0123456789'
      iarray(1) = 41
      iarray(2) = 42
      iarray(3) = 43

      WRITE (*, example)
```

The following output is produced:

```
&EXAMPLE
I1 =            11
I2 =            12
I4 =            14
L1 =  T
L2 =  T
L4 =  T
R4 =         24.000000
R8 =         28.000000000000000
Z8 =            (38.000000, 0.000000E+00)
Z16 =          (316.000000000000000, 0.000000000000000E+000)
C1 = 'A'
C10 = '0123456789'
IARRAY =            41           42           43
/
```

If a namelist-directed read is performed using `READ (*, example)` with the following input:

```
&example
I1=99
L1=F
R4=99.0
Z8=(99.0, 0.0)
C1='Z'
IARRAY(1)=99
/
```

a second `WRITE (*, example)` statement produces the following output:

```
&EXAMPLE
I1 =            99
I2 =            12
I4 =            14
L1 =  F
L2 =  T
L4 =  T
R4 =         99.000000
R8 =         28.000000000000000
Z8 =            (99.000000, 0.000000E+00)
Z16 =          (316.000000000000000, 0.000000000000000E+000)
C1 = 'Z'
C10 = '0123456789'
IARRAY =           99          42          43
/
```

ACTION Associates a unit number with an external file or device

SYNTAX

OPEN ([[**UNIT=**]*unitspec*
[[, **ACCESS=***access*]]
[[, **BLANK=***blanks*]]
[[, **BLOCKSIZE=***blocksize*]]
[[, **ERR=***errlabel*]]
[[, **FILE=***file*]]
[[, **FORM=***form*]]
[[, **IOFOCUS=***iofocus*]]
[[, **IOSTAT=***iocheck*]]
[[, **MODE=***mode*]]
[[, **RECL=***recl*]]
[[, **SHARE=***share*]]
[[, **STATUS=***status*]]
[[, **TITLE=***title*]])

If **UNIT=** is omitted, *unitspec* must be the first parameter. The parameters can otherwise appear in any order.

Parameter	Description
unitspec	An integer expression that specifies an external unit.
access	A character expression that evaluates to **'APPEND'**, **'DIRECT'**, or **'SEQUENTIAL'** (the default).
blanks	A character expression that evaluates to **'NULL'** or **'ZERO'**. Set to **'NULL'** (the default) to ignore blanks (same as **BN** edit descriptor, unless overridden by an explicit **BZ** in the **READ** statement). Set to **'ZERO'** to treat blanks as zeros (same as **BZ** edit descriptor, unless overridden by an explicit **BN** in the **READ** statement).
blocksize	An integer expression specifying internal buffer size for the unit (in bytes). See Section 3.2.4, "Input/Output Buffer Size." In a QuickWin application, *blocksize* sets the size of the screen buffer. The buffer size determines how much text is kept for the window. Setting the buffer to **'WINBUFINF'** continuously reallocates memory to save all the text. If *blocksize* is not specified, the buffer defaults to 2048 bytes.
errlabel	The label of an executable statement in the same program unit. If *errlabel* is specified, an I/O error causes transfer of control to the statement at *errlabel*. If *errlabel* is omitted, the effect of an I/O error is determined by the presence or absence of *iocheck*.

file

A character expression. If *file* is omitted, the compiler creates a scratch file unique to the specified unit. The scratch file is deleted when it is either explicitly closed or the program terminates normally.

If the file name is specified as a user file (**FILE=' USER'**) in a QuickWin application, a child window is opened. All subsequent **READ** and **WRITE** statements directed to that unit appear in the child window.

If the file name specified is blank (**FILE=' '**), the following steps are taken:

1. The program reads a file name from the list of arguments (if any) in the command line that invoked the program. If the argument is a null string (`' '`), the user is prompted for the corresponding file name. Each succeeding nameless **OPEN** statement reads a successive command-line argument.
2. If there are more nameless **OPEN** statements than command-line arguments, the program prompts for additional file names.

Assume the following command line has invoked the program MYPROG:

```
myprog first.fil " " third.txt
```

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 the file FIRST.FIL. Since a null argument was specified on the command line for the second file name, the **OPEN** statement for unit 4 produces the following prompt:

```
File name missing or blank -
Please enter name UNIT 4?
```

Unit 5 is associated with the file THIRD.TXT. As no fourth file was specified on the command line, the **OPEN** statement for unit 10 produces the following prompt:

```
File name missing or blank -
Please enter name UNIT 10?
```

If the file name is **'USER'**, input and output are directed to the console.

form

A character expression that evaluates to **'FORMATTED'**, **'UNFORMATTED'**, or **'BINARY'**.

If access is sequential, the default for *form* is **'FORMATTED** if access is direct, the default is **'UNFORMATTED'**.

iofocus

A logical expression specifying whether a new QuickWin child window will have the current I/O focus (i.e., is the active window). By default, *iofocus* is **.TRUE.**

iocheck

An integer variable, array element, or structure element that returns a value of zero if no error occurs, a negative integer if an end-of-file record is encountered, or the number of the error message if an error occurs. See Section 3.2.6, "Error and End-of-File Handling."

mode

A character expression that evaluates to **'READ'** (the process can only read the file), **'WRITE'** (the process can only write to the file), or **'READWRITE'** (the process can both read from and write to the file).

If *mode* is not specified, the FORTRAN run-time system attempts to open the file with a *mode* of **'READWRITE'**. If this open operation fails, the run-time system tries to open the file again, first using **'READ'**, then using **'WRITE'**.

Note that this action is not the same as specifying **MODE='READWRITE'**. If you specify the option **MODE='READWRITE'**, and the file cannot be opened for both read and write access, the attempt to open the file fails. The default behavior (trying **'READWRITE'**, then **'READ'**, then **'WRITE'**) is more flexible (though you may have to use the **INQUIRE** statement to determine the actual access mode selected). The value of the **STATUS=** option does not affect *mode*.

recl

An integer expression that specifies the length of each record in bytes. This parameter is required for direct-access files, and is ignored for sequential files.

share

A character expression. The acceptable values of *share* are:

Value	Description
'COMPAT'	Compatibility mode (the default mode in DOS). 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 (the default mode in OS/2). While a file is open in deny-none mode, any process may open the file in any mode (except compatibility mode). See Section 3.2.11 for information on file sharing.

status

A character expression that evaluates to **'OLD'**, **'NEW'**, **'UNKNOWN'** (default), or **'SCRATCH'**.

The following list describes the four values of *status*:

Value	Description
'OLD'	The file must already exist. If the file exists, it is opened. If it does not exist, an I/O error occurs. If you open an existing sequential file and write to it or close it without first moving to the end, you overwrite the file. (Opening with **ACCESS = 'APPEND'** automatically positions the file at the end.)
'NEW'	The file must not already exist. If the file does not exist, it is created. If it exists, an I/O error occurs. If you open a file with the option **STATUS='NEW'**, and close it with **STATUS='KEEP'**, or if the program terminates without explicitly closing the file, a permanent file is created.
'SCRATCH'	If you omit the *fname* parameter when opening a file, the value of *status* defaults to **'SCRATCH'**. Scratch files are temporary. They are deleted when explicitly closed or when the program terminates.
'UNKNOWN'	If you open a file with the **STATUS='UNKNOWN'** option, the FORTRAN run-time system first attempts to open the file with *status* equal to **'OLD'**, then with *status* equal to **'NEW'**. Therefore, if the file exists, it is opened; if it does not exist, it is created. The **'UNKNOWN'** status avoids any run-time errors associated with opening an existing file with **STATUS='NEW'** or opening a non-existent file with **STATUS='OLD'**.

Status values affect only disk files. When devices such as the keyboard or the printer are opened, the **STATUS=** option is ignored.

title — A character expression that evaluates to a string, which appears in the title bar of a child window in a QuickWin application. The default title string identifies the window as a unit and gives its *unitspec* number (that is, **TITLE='UNIT** *n*' where *n* is the unit's number).

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 a file.

If you open a file using a unit number that is already connected to another file, the already-open file is automatically closed first, then the new file name is connected to the unit; no error occurs. However, you cannot connect a single file to more than one unit at a time.

If you have not explicitly opened a file, and the first operation is a **READ** or **WRITE**, the FORTRAN run-time system attempts to open a file as if a blank name were specified, as described with the *file* parameter.

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 3.2.11, "File Sharing."

EXAMPLES

The following example opens a new file:

```
C     Prompt user for a file name and read it:
      CHARACTER*64 filename
      WRITE (*, '(A\)') ' enter file name '
      READ  (*, '(A)') filename
C     Open the file for formatted sequential access as unit 7.
C     Note that the specified access need not have been specified,
C     since it is the default (as is "formatted").
      OPEN (7, FILE = filename, ACCESS = 'SEQUENTIAL',
     +      STATUS = 'NEW')
```

The following example opens an existing file:

```
C     Open a file created by an editor, "DATA3.TXT", as unit 3:
      OPEN (3, FILE = 'DATA3.TXT')
```

ACTION Gives a constant a symbolic name

SYNTAX **PARAMETER** (*name=expression* [[, *name=expression*]]...)

Parameter	Description
name	A symbolic name.
expression	An expression. The *expression* can include symbolic names only if those names were defined in a previous **PARAMETER** statement in the program unit. If the expression is a character constant, it may contain no more than 1,957 bytes.

REMARKS If *expression* is of a different numerical type than *name*, any needed conversion is performed automatically.

Unlike the **DATA** statement (which declares an initial value for a variable), the **PARAMETER** statement sets fixed values for symbolic constants. Any attempt to change a symbolic constant with an assignment statement or a **READ** statement causes a compile-time error.

A symbolic constant cannot be of a structure type, and it may not appear in a format specification.

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*(*) bigone
      PARAMETER (bigone = 'This constant is larger than 40
     +characters')
```

ACTION Temporarily suspends program execution and allows you to execute operating system commands during the suspension

SYNTAX **PAUSE** [[*prompt*]]

Parameter	Description
prompt	Either a character constant or an integer from 0 to 99,999

REMARKS If *prompt* is not specified, the following default prompt appears:

```
'Please enter a blank line (to continue) or a system command.'
```

If *prompt* is a number, the number appears on the screen preceded by `PAUSE -` . If *prompt* is a string, only the string is displayed.

The following are possible responses to the prompt and their results:

If the user enters:	Then:
A blank line	Control returns to the program.
A command	The command is executed and control returns to the program. (The maximum size for a command specified on one line is 128 bytes.)
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). (Under OS/2, the user must enter "CMD".)

EXAMPLES

```
SUBROUTINE SetDrive ( )
   PAUSE 'Please select default drive.'
```

One of the following outputs is produced:

```
Please select default drive.
.
.
.

PAUSE 999.
.
.
.

PAUSE- 999
```

ACTION Displays output on the screen

SYNTAX **PRINT** { *, | *formatspec* | *namelist* } [[, *iolist*]]

Parameter	Description
formatspec	A format specifier
namelist	A namelist specifier
iolist	A list specifying the data to be transferred

REMARKS If *, replaces *formatspec*, output is list-directed.

If a namelist specifier is used, no *iolist* may appear.

The **PRINT** statement writes only to unit*.

The iolist may not contain structure variables, only structure elements.

If a parameter of the **PRINT** statement is an expression that calls a function, that function must not execute an I/O statement or the **EOF** intrinsic function, because the results are unpredictable.

EXAMPLE

```
C     The two following statements are equivalent:
      PRINT     '(A11)', 'Abbottsford'
      WRITE (*, '(A11)') 'Abbottsford'
```

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

SYNTAX **PROGRAM** *program-name*

Parameter	Description
program-name	The name you have given to your main program. The program name is global. Therefore, it cannot be the name of a user-defined procedure or common block. It is also a local name and must not conflict with any name local to the main program.

REMARKS The **PROGRAM** statement may only appear as the first statement of a main program.

The main program is always assigned the system name of **_main**, in addition to any name specified by the **PROGRAM** statement.

EXAMPLE

```
PROGRAM Gauss
.
.
.
END
```

ACTION

Transfers data from the file associated with *unitspec* to the items in the *iolist* until all read operations have been completed, the end-of-file is reached, or an error occurs

SYNTAX

READ { { *formatspec*,| *nmlspec* } |
(**[[UNIT=]]***unitspec*
[[, **[[**{**[[FMT=]]** *formatspec***]]** | **[[NML=]]***nmlspec***}]]**
[[, **END=***endlabel***]]**
[[, **ERR=***errlabel***]]**
[[, **IOSTAT=***iocheck***]]**
[[, **REC=***rec***]])}**
iolist

If **UNIT=** is omitted, *unitspec* must be the first parameter. If **FMT=**or **NML=** is omitted, *fmtspec* or *nmlspec* must be the second parameter. The parameters can otherwise appear in any order.

Parameter	Description
unitspec	A unit specifier. When reading from an internal file, *unitspec* is a character substring, character variable, character array element, character array,noncharacter array, or structure element . When reading from an external file, *unitspec* is an integer expression that specifies a unit . If *unitspec* has not been explicitly associated with a file, an "implicit open," using the following syntax is performed: **OPEN** (*unitspec*, **FILE** = ' ', **STATUS** = '**OLD**', **ACCESS** = '**SEQUENTIAL**', **FORM** = *form*) where *form* is **'FORMATTED'** for a formatted read operation or **'UNFORMATTED'** for an unformatted read operation. If you included a file name on the command line that executed this program, that name will be used for the file. If you did not include a file name, you will be prompted for one. For more information, see Chapter 3, "The Input/Output System."

formatspec

A format specifier. It can be the label of a **FORMAT** statement, or the format specification itself.

The *formatspec* argument is required for a formatted read, and must not appear in an unformatted read. 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).

nmlspec

A namelist group specifier. If given, *iolist* must be omitted. Namelist-directed I/O may be performed only on files opened for sequential access.

endlabel

The label of a statement in the same program unit.

If *endlabel* is present, reading the end-of-file record transfers control to the executable statement at *endlabel*. If *endlabel* is omitted, reading past the end-of-file record causes a run-time error, unless either *errlabel* or *iocheck* is specified.

errlabel

The label of an executable statement in the same program unit.

If *errlabel* is specified, an I/O error causes transfer of control to the statement at *errlabel*. If *errlabel* is omitted, the effect of an I/O error is determined by the presence or absence of *iocheck*.

iocheck

An integer variable, array element, or structure element that returns a value of zero if there is no error, –1 if end-of-file is encountered, or the number of the error message if an error occurs. For more information on error handling, see Section 3.2.6, "Error and End-of-File Handling."

rec

An integer expression that evaluates to a positive number, called the record number. It is specified for direct-access files only. If *rec* is specified for a sequential-access or an internal file, an I/O error occurs.

The file is positioned to record number *rec* before data is read from it. (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 into which values are transferred from the file. If I/O is to or from a formatted device, an *iolist* may not contain structure variables, but it may contain structure elements.

REMARKS

If the file were opened with **MODE='READWRITE'** (the default), you could read and write to the same file without having to reopen it to reset the access mode.

If you attempt to read an unwritten record that has not been written to from a direct file, the results are undefined.

If a parameter of the **READ** statement is an expression that calls a function, that function must not execute an I/O statement or the **EOF** intrinsic function, because the results are unpredictable.

EXAMPLE

```
C     Define a two-dimensional array:
      DIMENSION ia(10,20)
C     Read in the bounds for the array. These bounds should
C     be less than or equal to 10 and 20, respectively.
C     Then read in the array in nested implied-DO lists
C     with input format of 8 columns of width 5 each.
      READ   (3, 990) il, jl, ((ia(i,j), j = 1, jl), i =1, il)
  990 FORMAT (2I5, /, (8I5))
```

ACTION Specifies the **REAL** type for 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. Specifies the length, in *bytes*, of the names in 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, intrinsic function, **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, ALLOCATABLE, C, EXTERN, FAR, HUGE, NEAR, PASCAL, REFERENCE, VALUE.**
length	Must be 4 or 8. Gives *vname* the specified *length*. The *length* parameter overrides the length specified by *bytes*.
dim	A dimension declarator. Specifying *dim* declares *vname* as an array.
values	A list of constants and repeated constants, separated by commas. A repeated constant takes the form *n***constant*, where *n* is a positive integer constant. The */values/* option, if specified, initializes *vname*. The following statement declares that `num` is of type **REAL**, and sets `num` equal to `10.0`:

```
REAL num /10.0/
```

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.

If both the *bytes* and *length* parameters are omitted, the default length is four bytes.

REAL statements must precede all executable statements.

REAL*8 and **DOUBLE PRECISION** are the same data type.

EXAMPLES

```
REAL goof, abs
REAL*4 wx1, wx3*8, wx5, wx6*8
```

ACTION Specifies a user-defined structure type for user-defined names

SYNTAX **RECORD** */type_name/ vname* [[*[attrs]*]] [[*(dim)*]]
[[, *vname*[[*[attrs]*]] [[*(dim)*]]]...

Parameter	Description
type_name	The name of user-defined structure type.
vname	A variable, array, or array declarator, which will be of this structure type.
attrs	A list of attributes, separated by commas. The *attrs* describe *vname*.
dim	A dimension declarator. Specifying *dim* declares *vname* as an array.

REMARKS The **STRUCTURE** statement defines a new data type. The **RECORD** statement assigns this new data type to a variable. The *type-name* must be defined prior to its appearance in a **RECORD** statement.

RECORD statements must precede all executable statements.

EXAMPLE

```
STRUCTURE /address/
  LOGICAL*2    house_or_apt
  INTEGER*2    apt
  INTEGER*2    housenumber
  CHARACTER*30 street
  CHARACTER*20 city
  CHARACTER*2  state
  INTEGER*4    zip
END STRUCTURE

RECORD /address/ mailing_addr[NEAR](20), shipping_addr(20)
```

SEE ALSO **STRUCTURE**

ACTION Returns control to the calling program unit

SYNTAX **RETURN** [[*ordinal*]]

Parameter	Description
ordinal	An integer constant. Specifies the position of an alternate-return label in the formal argument list for the subroutine. See the **CALL** entry in this section for information on alternate returns.

REMARKS The **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. The **END** statement in a function or subroutine has the same effect as 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 via *ordinal*. The **RETURN** *ordinal* statement must appear only in a subroutine; it may not appear in a function.

A **RETURN** statement in the main program is treated as a **STOP** statement with no *message* parameter.

EXAMPLES

```
C     This subroutine loops until you type "Y":
      SUBROUTINE Loop
      CHARACTER in

   10 READ (*, '(A)') in
      IF (in .EQ. 'Y') RETURN
      GOTO 10
C     RETURN implied by the following statement:
      END
```

The following example demonstrates the alternate-return feature:

```
   01 CALL AltRet (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 AltRet (i, *, j, *, *)
IF (i .EQ. 10)  RETURN 1
IF (i .EQ. 20)  RETURN 2
IF (i .EQ. 30)  RETURN 3
RETURN
```

In this example, `RETURN 1` specifies the list's first alternate-return label, which is a symbol for the actual argument `*10` in the `CALL` statement. `RETURN 2` specifies the second alternate-return label, and `RETURN 3` specifies the third alternate-return label.

ACTION Repositions a file to its first record

SYNTAX

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

If **UNIT=** is omitted, *unitspec* must be the first parameter. The parameters can otherwise appear in any order.

Parameter	Description
unitspec	An integer expression that specifies an external unit. If *unitspec* has not been opened, **REWIND** has no effect.
errlabel	The label of an executable statement. If *errlabel* is specified, an I/O error causes transfer of control to the statement at *errlabel*. If *errlabel* is omitted, the effect of an I/O error is determined by the presence or absence of *iocheck*.
iocheck	An integer variable, array element, or structure element that returns a value of zero if there is no I/O error, or the number of the error message if an error is encountered. See Section 3.2.6, "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 execute an I/O statement or the **EOF** intrinsic function, because the results are unpredictable.

EXAMPLE

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

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. If no names are specified, all allowable items in the program unit are saved.

REMARKS The **SAVE** statement does not allow the following:

- Repeated appearances of an item
- Formal argument names
- Procedure names
- Names of entities in a common block
- Names that appear in an **AUTOMATIC** statement

Within a program, a common block that is saved in one subprogram must be saved in every subprogram containing that common block.

NOTE *Microsoft FORTRAN saves all variables by default. The* ***SAVE*** *statement simplifies porting code, and can specify variables that are not to be made automatic.*

EXAMPLE

```
SAVE /mycom/, myvar
```

SEE ALSO **AUTOMATIC**

ACTION

Transfers program control to a selected block of statements, determined by the value of a controlling expression

SYNTAX

SELECT CASE (*testexpr*)

CASE (*expressionlist*)
[[*statementblock*]]

[[**CASE** *(expressionlist)*
[[*statementblock*]]]]
.
.
.
[[**CASE DEFAULT**
[[*statementblock*]]]]

END SELECT

Parameter	Description
testexpr	An **INTEGER**, **LOGICAL**, or **CHARACTER*1** expression.
expressionlist	A list of values, which must be constant and must match the data type of *testexpr*. The values must be of type **INTEGER**, **LOGICAL**, or **CHARACTER*1**. If *testexpr* matches one of the values, the following *statementblock* is executed.
statementblock	One or more executable statements.

REMARKS

There are two ways to include values in the *expressionlist*. The first is to give a list of individual values, separated by commas. The second is to specify an inclusive range of values, separated by a colon, such as `5:10` or `'I':'N'`. The lower bound must be less than the upper bound. If the values are characters, the first character must appear before the second in the ASCII collating sequence.

If the lower bound is omitted (such as `:10`), then all values less than or equal to the upper bound match. If the upper bound is omitted (such as `5:`), then all values greater than or equal to the lower bound must match. Both individual values and ranges of values may be included in the same *expressionlist*. You cannot specify a range of values when *testexpr* is of type **LOGICAL**. A value (even when specified implicitly as part of a colon range) can only appear in one *expressionlist*.

A *statementblock* need not contain executable statements. Empty blocks can be used to make it clear that no action is to be taken for a particular set of expression values.

The **CASE DEFAULT** statement is optional. You can include only one **CASE DEFAULT** statement in a **SELECT CASE** block.

If the value of *testexpr* does not match any value in any *expressionlist*, execution passes beyond the **SELECT CASE** construct to the next executable statement.

SELECT CASE blocks may be nested. Each block must have its own **END SELECT** statement; you may not terminate more than one block with a single **END SELECT** statement.

If a **SELECT CASE** statement appears within a **DO** or **DO WHILE** loop, or within an **IF**, **ELSE**, or **ELSE IF** structure, the terminating **END SELECT** statement must appear within the range of that loop or structure.

It is illegal to branch into a **SELECT CASE** block from outside, or to branch from one **CASE** section to another. Any attempt to do so causes a compile-time error.

EXAMPLE

```
CHARACTER*1  cmdchar
.
.
.
SELECT CASE (cmdchar)
   CASE ('0')
      WRITE (*, *) "Must retrieve one to nine files"
   CASE ('1':'9')
      CALL RetrieveNumFiles (cmdchar)
   CASE ('A', 'd')
       CALL AddEntry
   CASE ('D', 'd')
      CALL DeleteEntry
   CASE ('H', 'h')
      CALL Help
   CASE ('R':'T', 'r':'t')
      WRITE (*, *) "REDUCE, SPREAD and TRANSFER commands ",
     +    "not yet supported"
   CASE DEFAULT
      WRITE (*, *) "Command not recognized; please
     +re-enter"
END SELECT
```

ACTION Defines a function in a single 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 arguments. If there is more than one name, the names must be separated by commas. 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, a reference to *formal* within the statement function always refers to the formal argument, never to the other local name. The data type of *formal* is determined by the data type of the local variable, because the compiler has no other way to establish a type for it.
expression	Any legal expression. References to variables, formal arguments, other functions, array elements, and constants are permitted within the *expression*. 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 Like a regular function, a statement function is executed by referring to it 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. 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 be an array definition) and in a **COMMON** statement (as the name of a common block).

EXAMPLE

```
      DIMENSION x(10)
      Add (a, b) = a + b

      DO 100, n = 1, 10
      x(n) = Add (y, z)
100   CONTINUE
```

ACTION Terminates program execution

SYNTAX **STOP** [*message*]

Parameter	Description
message	A character constant or an integer constant from 0 to 99,999

REMARKS If no *message* is specified, the following default message is displayed:

```
STOP - Program terminated.
```

If *message* is a character constant, it is displayed, and the program returns zero to the operating system, for use by program units that retrieve status information.

If *message* is a number:

1. The words `Return code` , followed by the number, are displayed. For example, the statement `STOP 0400` produces the 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, for use by program units that wish to check status.

EXAMPLE

```
      IF (ierror .EQ. 0) GOTO 200
      STOP 'ERROR DETECTED!'
  200 CONTINUE
```

Stop NOT ALLOWED IN DLL's

ACTION Defines a new, compound variable type constructed from other variable types

SYNTAX

STRUCTURE */type_name/*
element_declaration(s)
.
.
.
END STRUCTURE

Parameter	Description
type_name	The name for the new data type; follows standard naming conventions. It may not duplicate the name of another variable or an intrinsic function, and it may not be the name of an intrinsic data type, such as **COMPLEX**. It is a local name.
element_declarations	Any combination of one or more variable-typing statements and **UNION** constructs. Can include **RECORD** statements that use previously defined structure types.

REMARKS The **STRUCTURE** statement defines a new variable type, called a "structure"; it is not a declaration of a specific program variable. To declare a variable of a particular structure type, use the **RECORD** statement.

A structure is made of elements. The simplest element is a conventional FORTRAN variable type with a dummy name, as in the following example:

```
STRUCTURE /full_name/
   CHARACTER*15 first_name
   CHARACTER*20 last_name
END STRUCTURE
```

Elements cannot be declared with attributes. For example, `INTEGER var[FAR]` is not permitted.

An element can also be a **RECORD** statement that references a previously defined structure type:

```
STRUCTURE /full_address/
   RECORD /full_name/ personsname
   RECORD /address/   ship_to
   INTEGER*1          age
   INTEGER*4          phone
END STRUCTURE
```

An element can also be a union of several variable maps.

Element names are local to the structure in which they appear. There is no conflict if the same element name appears in more than one structure. Nested structures may have elements with the same names.

A particular element is specified by listing the sequence of elements required to reach the desired element, separated by a period (.). Suppose a structure variable, *shippingaddress*, were declared with the *full_address* structure declared in the previous example:

```
RECORD /full_address/ shippingaddress
```

The age element would then be specified by `shippingaddress.age`, the first name of the receiver by `shippingaddress.personsname.first_name`, and so on. An element is no different from any other variable having the same type except that it cannot be used as a loop counter. When a structure element appears in an expression, its type is the type of the element. When passed as an actual argument, it must match the formal argument in type, order, and dimension.

A structure can be no longer than 65,536 bytes. This includes all the data, plus any padding bytes. The way structures are packed in memory is controlled by the **$PACK** metacommand and /Zp command-line option.

Structures may look identical, but still be different. For two structures to be identical, they must have the same component types and names in the same sequence, and the packing must be the same.

SEE ALSO

MAP...END MAP
RECORD
UNION...END UNION
$PACK

ACTION

Identifies a program unit as a subroutine, gives it a name, and identifies its formal arguments

SYNTAX

SUBROUTINE *subr* [[*sattrs*]] [([[*formal* [[*attrs*]] [[,*formal* [[*attrs*]]]...])]]

Parameter	Description
subr	The global, external name of the subroutine. The name *subr* cannot appear in **AUTOMATIC**, **COMMON**, **DATA**, **EQUIVALENCE**, **INTRINSIC**, **LOADDS**, **SAVE**, or type statements.
sattrs	A list of attributes, separated by commas. The *sattrs* describes *subr*. The following attributes can be used with *subr*: **ALIAS**, **C**, **FAR**, **NEAR**, **PASCAL**, **VARYING**.
formal	A formal-argument name. If more than one is specified, they must be separated by commas. A formal argument may be the name of a conventional variable, a structure variable, or an intrinsic function. A formal argument may also be an alternate return label (*). For an explanation of alternate return specifiers, see the **CALL** entry in this section.
attrs	A list of attributes, separated by commas. The *attrs* describes *formal* (except alternate-return specifiers). 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. A subroutine can contain any statement except a **BLOCK DATA**, **FUNCTION**, **INTERFACE TO**, **PROGRAM**, or **SUBROUTINE** statement.

Within the calling program, *subr* is global, and may not be used for any other variable or subprogram.

The formal-argument list sets the number of arguments for that subroutine. The argument types are set by any **IMPLICIT**, **EXTERNAL**, **DIMENSION**, or type statements within the subroutine itself (or they default to implicit FORTRAN types). The argument types are not set by the calling program, even if the subroutine appears in the calling program's source file. Formal-argument names cannot appear in **AUTOMATIC**, **COMMON**, **DATA**, **EQUIVALENCE**, **INTRINSIC**, or **SAVE** statements.

In a **CALL** statement, the actual arguments that are passed 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 checks for correspondence if the formal arguments are known.

A "recursive" subroutine is one that calls itself, or calls another subprogram which in turn calls the first subroutine before the first subroutine has completed execution. FORTRAN does not support recursive subroutine calls. When using Microsoft FORTRAN, any attempt at direct recursion results in a compile-time error. Indirect recursion, however, is not detected. The results of such indirect recursion are undefined and unpredictable.

EXAMPLE

```
      SUBROUTINE GetNum (num, unit)
      INTEGER num, unit
10    READ (unit, '(I10)', ERR = 10) num
      END
```

See the individual listings in this chapter for the **CHARACTER**, **COMPLEX**, **DOUBLE COMPLEX**, **DOUBLE PRECISION**, **INTEGER**, **LOGICAL**, **REAL**, **RECORD**, and **STRUCTURE...END STRUCTURE** statements.

ACTION

Causes two or more maps to occupy the same memory location

SYNTAX

UNION
map-statement
map-statement
[[*map-statement*]] ...
END UNION

Parameter	Description
map-statement	A map declaration. See the **MAP..END MAP** entry in this section for more information.

REMARKS

A **UNION** block may only appear within a **STRUCTURE** block. Each **UNION** block must be terminated with its own **END UNION** statement; a single **END UNION** statement may not be used to terminate more than one **UNION** block.

UNION is similar to the **EQUIVALENCE** statement; both allocate the same memory area to more than one variable. However, maps can contain multiple, contiguous variables, which gives increased flexibility in assigning variables to the same memory location.

EXAMPLE

Note how the first 40 characters in the `string2` array are assigned to four-byte integers, while the remaining 20 are assigned to two-byte integers:

```
      UNION
         MAP
            CHARACTER*20 string1, CHARACTER*10 string2(6)
         END MAP
         MAP
            INTEGER*2 number(10), INTEGER*4 var(10), INTEGER*2
     +      datum(10)
         END MAP
      END UNION
```

The **$PACK** metacommand and /Zp command-line option control how variables in structures are assigned to beginning byte addresses. The particular packing option chosen may affect how particular variables are assigned to the same memory location. See the **$PACK** entry in Section 6.2, "Metacommand Directory," for more information.

SEE ALSO

MAP...END MAP
STRUCTURE...END STRUCTURE
$PACK

ACTION Transfers data from the *iolist* items to the file associated with *unitspec*

SYNTAX

WRITE ([[**UNIT=**]]*unitspec*
[[, [[{[[**FMT=**]]*formatspec*]]|[[**NML=**]]*nmlspec* }]]
[[, **ERR**=*errlabel*]]
[[, **IOSTAT**=*iocheck*]]
[[, **REC**=*rec*]])
iolist

If **UNIT=** is omitted, *unitspec* must be the first parameter. If **FMT=** or **NML=** is omitted, *fmtspec* or *nmlspec* must be the second parameter. The parameters can otherwise appear in any order.

Parameter	Description
unitspec	When writing to an internal file, *unitspec* must be a character substring, variable, array, array element, structure element, or noncharacter array. When writing to an external file, *unitspec* is an integer expression that specifies a unit. If *unitspec* has not been explicitly associated with a file, an "implicit open," using the following syntax, is performed: **OPEN** (*unitspec*, **FILE** = ' ', **STATUS** = '**UNKNOWN**', **ACCESS** = '**SEQUENTIAL**', **FORM** = *form*) where *form* is **'FORMATTED'** for a formatted write operation and **'UNFORMATTED'** for an unformatted write operation.
formatspec	A format specifier. A format specifier is required for a formatted write; it must not appear in an unformatted write.
nmlspec	A namelist specifier. If specified, *iolist* must be omitted. Namelist-directed I/O may be performed only on files opened for sequential access.
errlabel	The label of an executable statement in the same program unit. If *errlabel* is specified, an I/O error causes transfer of control to the statement at *errlabel*. If *errlabel* is omitted, the effect of an I/O error is determined by the presence or absence of *iocheck*.

iocheck — An integer variable, array element, or structure element that returns a value of zero if there is no error, or the number of the error message if an error occurs. See Section 3.2.6, "Error and End-of-File Handling."

rec — A positive integer expression, called a record number, specified only for direct-access files (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. It may not contain structure variables, though structure elements are permitted.

REMARKS

If the file were opened with **MODE='READWRITE'** (the default), you could alternately read and write to the same file without reopening it each time.

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.

If a parameter of the **WRITE** statement is an expression that calls a function, that function must not execute an I/O statement or the **EOF** intrinsic function, 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) (n, SQRT(FLOAT(n)), FLOAT(n)**(1.0/3.0),
      +n = 1, 100)
      10 FORMAT (I5, F8.4, F8.5)
```

CHAPTER 5

5

Intrinsic Functions and Additional Procedures

Intrinsic functions are predefined by the Microsoft FORTRAN language. These functions carry out data-type conversions and return information about data types, perform operations on both numeric and character data, test for end-of-file, return addresses, and perform bit manipulation.

The first part of this chapter is a detailed description of the intrinsic functions available in Microsoft FORTRAN. The second part of this chapter is an alphabetical tabular listing of all intrinsic functions. The third part of this chapter describes other functions and subroutines supplied with Microsoft FORTRAN.

5.1 Using Intrinsic Functions

An "intrinsic function" is a function that is part of the FORTRAN language. The compiler recognizes its name and knows that the object code for the function is in the language libraries. Intrinsic functions are automatically linked to the program without any additional effort on the programmer's part. (Functions written by the user are called "external" functions. If you wish to write your own function that has the same name as an intrinsic function, you must declare its name in an **EXTERNAL** statement.)

Each function returns a value of integer, real, logical, or character, and therefore has a corresponding data type. For example, the **ICHAR** intrinsic function returns the ASCII value of a character string; it is therefore an integer function. Some intrinsic functions, such as **INT**, can take arguments of more than one type. Others, such as **ABS**, can return a value that has the same type as the argument.

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 function call should be of the same type. The intrinsic function **DIM**, for example, takes two integer arguments and returns the positive difference. If you specify `DIM (i,j)`, `i` and `j` must be of the same type.

If, however, two arguments are of different types, Microsoft FORTRAN first attempts to convert the arguments to the correct data type. For example, if `i` and `j` in the example above are real numbers, they are first converted to **INTEGER** type (see Table 1.6, "Arithmetic Type Conversion," for information on type conversion), and then the operation is performed. 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 is performed.

Intrinsic-function names, except those listed in Section 2.5, "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.

Generic intrinsic functions, like **ABS**, let you 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.

An external procedure cannot have the same name as an intrinsic function, unless the **EXTERNAL** statement is used to tell the compiler the substitution has been made. (As a result, the intrinsic function of that name is no longer usable in the program unit where the **EXTERNAL** statement associated it with an external procedure.) In the following statement, for example, `sign`, `float`, and `index` are the names of external procedures:

```
EXTERNAL Sign, Float, Index
SUBROUTINE Process (Sign, Float, Index)
```

If you supply an argument that has no mathematically defined result or for which the result exceeds the numeric range of the processor, the result of the intrinsic function is undefined.

Arguments must agree in order, number, and type with those specified in Tables 5.2–5.17. Arguments can also be expressions of the specified type.

When logarithmic and trigonometric intrinsic functions act on a complex argument, they return the "principal value." The principal value of a complex number is the number whose argument (angle in radians) is less than or equal to π and greater than $-\pi$.

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

WARNING *Microsoft FORTRAN contains several intrinsic functions whose names are longer than six characters (**ALLOCATED, EPSILON, LEN_TRIM, LOCNEAR, MAXEXPONENT, MINEXPONENT, NEAREST, PRECISION**). If the **$TRUNCATE** metacommand is enabled, the compiler considers any function name with the same first six characters to be one of these intrinsic functions.*

Table 5.1 lists the abbreviations used in the tables of intrinsic functions in this chapter.

Table 5.1 Abbreviations Used to Describe Intrinsic Functions

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

5.1.1 Data-Type Conversion

This section describes the type-conversion intrinsic functions. Table 5.2 summarizes the intrinsic functions that perform type conversion.

Table 5.2 Intrinsic Functions: Type Conversion[1]

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*)	**DOUBLE PRECISION, REAL*8**	int
REAL (*gen*)	int, real, or cmp	**REAL*4**
DREAL (*cmp*)	**COMPLEX*16**	**REAL*8**
FLOAT (*int*)	int	**REAL*4**
SNGL (*dbl*)	**REAL*8**	**REAL*4**
DBLE (*gen*)	int, real, or cmp	**DOUBLE PRECISION**
DFLOAT (*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

[1] Note: See Table 5.1 for a list of 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*) is the truncated value of *gen*. `INT (1.9)`, for example, is equal to 1, and `INT (-1.9)` is equal to –1. If *gen* is complex, the real part of *gen* is taken, then truncated to an integer.

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 an 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 1.6.3, "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, **REAL**(*gen*) is *gen* stored as a single-precision real number. If *gen* is a single-precision real number, **REAL**(*gen*) equals *gen*. If *gen* is complex, **REAL**(*gen*) equals the real part of *gen*. If *gen* is a double-precision number, **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, a real, a double-precision, or a complex number. If *gen* is complex, **CMPLX**(*gen*) equals *gen*. If *gen* is an integer, a real, or a 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 have 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 ASCII characters into integers, and the **CHAR** intrinsic function translates integers into ASCII characters. (The ASCII character set is listed 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.

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`

5.1.2 Data-Type Information

This section describes the intrinsic functions that return information about data types. Except for the **NEAREST** function, the variables passed as arguments do not need to have been assigned a value; it is the data type of the argument that is significant. Table 5.3 summarizes these functions.

Table 5.3 Intrinsic Functions: Data-Type Information[1]

Name	Argument Type	Function Type
ALLOCATED (*gen*)	array	log
EPSILON (*real*)	real	real
HUGE (*gen*)	int or real	Same as argument
MAXEXPONENT (*real*)	real	real
MINEXPONENT (*real*)	real	real
NEAREST (*real, director*)	real	real
PRECISION (*real*)	real	real
TINY (*real*)	real	real

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

The **ALLOCATED** intrinsic function returns a logical value that is **.TRUE.** if memory is currently allocated to the array.

The **EPSILON** intrinsic function returns the smallest increment that, when added to one, produces a number greater than one for the argument's data type. The increment is slightly larger than the best precision attainable for that data type. For example, a single-precision real can accurately represent some numbers to seven decimal places: **EPSILON** (*real4*) returns 1.112093E-07.

The **EPSILON** function makes it easy to select a *delta* for algorithms (such as root locators) that search until the calculation is within *delta* of an estimate. If *delta* is too small (smaller than the decimal resolution of the data type), the algorithm may never halt. By scaling the value returned by **EPSILON** to the estimate, you obtain a *delta* that ensures search termination.

The **HUGE** intrinsic function returns the largest positive number that can be represented by the argument's data type.

The **MAXEXPONENT** intrinsic function returns the largest positive decimal exponent that a number of the argument's data type can have. For example, **MAXEXPONENT** (*real4*) returns 38.

The **MINEXPONENT** intrinsic function returns the largest negative decimal exponent that a number of the argument's data type can have. For example, **MINEXPONENT** (*real8*) returns –308.

The **NEAREST** intrinsic function returns the nearest different number in the direction of *director*. (If *director* is positive, the value returned is larger than *real*. If *director* is negative, the value returned is smaller than *real*.) As with the **EPSILON** intrinsic function, **NEAREST** lets the program select an appropriate increment to guarantee a search terminates.

The **PRECISION** intrinsic function returns the number of significant decimal digits for *real*'s data type. It is useful in rounding off numbers.

The **TINY** intrinsic function returns the smallest positive number that can be represented by the argument's data type.

5.1.3 Truncating and Rounding

Table 5.4 summarizes the intrinsic functions that truncate and round.

Table 5.4 Intrinsic Functions: Truncation and Rounding[1]

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

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

The intrinsic functions **AINT** and **DINT** truncate their arguments. The intrinsic functions **ANINT**, **DNINT**, **NINT**, and **IDNINT** round their arguments and 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(2.6)	2.0
AINT(-2.6)	-2.0
ANINT(2.6)	3.0
ANINT(-2.6)	-3.0

The following example 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 = 0.081
      amount  = 12.99
C     Calculate tax and round to nearest hundredth:
      tax = ANINT (amount * taxrate * 100.0) / 100.0
      total = amount + tax
      WRITE (*, 100) amount, tax, total
  100 FORMAT (1X, 'AMOUNT', F7.2 /
     +        1X, 'TAX   ', F7.2 /
     +        1X, 'TOTAL ', F7.2)
      END
```

5.1.4 Absolute Value and Sign Transfer

Table 5.5 summarizes the intrinsic functions that compute absolute values and perform sign transfers.

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:

$$\sqrt{real^2 + imag^2}$$

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.

Table 5.5 Intrinsic Functions: Absolute Values and Sign Transfer[1]

Name	Definition	Argument Type	Function Type
ABS (*gen*)	Absolute value	int, real, or cmp	Function type same as argument type, except when argument is complex[2]
IABS (*int*)	Absolute value	int	int
DABS (*dbl*)	Absolute value	**REAL*8**	**REAL*8**
CABS (*cmp*)	Absolute value	cmp	real[2]
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**

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

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

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 is -5.2:
      WRITE (*, *) c
      END
```

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

```
C     This program takes the square root of a vector
C     magnitude. Since the sign in a vector represents
C     direction, the square root of a negative value is not
C     meant to produce complex results. This routine removes
C     the sign, takes the square root, and then restores
C     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 (1.0, mag)

C     Calculate the square root of the absolute value
C     of the magnitude:
      result = SQRT (ABS (mag))

C     Restore the sign by multiplying the result by -1 or
C     +1:
      result = result * sgn
      WRITE (*, *) result
      END
```

5.1.5 Remainders

Table 5.6 summarizes the intrinsic functions that return remainders.

Table 5.6 Intrinsic Functions: Remainders[1]

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**

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

The intrinsic functions **MOD**, **AMOD**, and **DMOD** return remainders as follows (**AMOD** and **DMOD** use exactly the same formula as **MOD**):

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

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

5.1.6 Positive Differences

Table 5.7 summarizes the intrinsic functions that return the positive difference between two arguments.

Table 5.7 Intrinsic Functions: Positive Difference[1]

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**

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

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

Situation	Result
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

5.1.7 Maximums and Minimums

Table 5.8 summarizes the functions that return the maximum or minimum of two or more values.

Table 5.8 Intrinsic Functions: Maximums and Minimums[1]

Name	Definition	Argument Type	Function Type
MAX (*genA*, *genB*[[, *genC*]]...)	Maximum	int or real	Same as argument
MAX0 (*intA*, *intB*[[, *intC*]]...)	Maximum	int	int
AMAX1 (*real4A*, *real4B*[[, *real4C*]]...)	Maximum	**REAL*4**	**REAL*4**
AMAX0 (*intA*, *intB*[[, *intC*]]...)	Maximum	int	**REAL*4**

Table 5.8 *(continued)*

Name	Definition	Argument Type	Function Type
MAX1 (*real4A*, *real4B*[[, *real4C*]]...)	Maximum	**REAL*4**	int
DMAX1(*dblA*, *dblB*[[, *dblC*]]...)	Maximum	**REAL*8**	**REAL*8**
MIN (*genA*, *genB*[[, *genC*]]...)	Minimum	int or real	Same as argument
MIN0 (*intA*, *intB*[[, *intC*]]...)	Minimum	int	int
AMIN1 (*real4A*, *real4B*[[, *real4C*]]...)	Minimum	**REAL*4**	**REAL*4**
AMIN0 (*intA*, *intB*[[, *intC*]]...)	Minimum	int	**REAL*4**
MIN1 (*real4A*, *real4B*[[, *real4C*]]...)	Minimum	**REAL*4**	int
DMIN1 (*dblA*, *dblB*[[, *dblC*]]...)	Minimum	**REAL*8**	**REAL*8**

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

The intrinsic functions **MAX**, **MAX0**, **AMAX1**, and **DMAX1** return the maximum value in the argument list. The intrinsic functions **AMAX0** and **MAX1** return the maximum and also perform type conversion. Similarly, **MIN**, **MIN0**, **AMIN1**, and **DMIN1** return minimums, while **AMIN0** and **MIN1** return the minimum and also perform type conversion.

Examples

The following list shows examples of maximums and minimums:

Function Reference	Equivalent
`MAX(5,6,7,8)`	`8`
`MAX(-5.,-6.,-7.)`	`-5.`
`MIN(-5,-6,-7)`	`-7`
`MIN(.1E12,.1E14,.1E19)`	`.1E12`

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

```
C     This program uses the MAX intrinsic function to find
C     the 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, 2.0, -6.3,
     +        2.5, -12.2/
```

```
C     Initialize small and large with arbitrarily large and
C     small 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/
     +        ' The largest number was  ', F6.1)
      END
```

Output

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

5.1.8 Double-Precision Products

Table 5.9 lists the intrinsic function that returns a double-precision product.

Table 5.9 Intrinsic Functions: Double-Precision Product[1]

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

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

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

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

```
      REAL  a, b

      a = 3.72382
      b = 2.39265

      WRITE (*, *) a*b, DPROD (a,b)
```

The following output is produced:

```
8.9097980          8.90979744044290
```

5.1.9 Complex Functions

Table 5.10 lists the intrinsic operations that perform various other operations on complex numbers.

Table 5.10 Intrinsic Functions: Complex Operators[1]

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**

[1] Note: See Table 5.1 for a list of 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. Therefore, 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 5.1.1, 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**2 - 4.0*a*c))
      ans1 = (-b + desc) / (2.0 * a)
      ans2 = (-b - desc) / (2.0 * 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')
      END
```

5.1.10 Square Roots

Table 5.11 summarizes the intrinsic functions that return square roots.

Table 5.11 Intrinsic Functions: Square Roots[1]

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**

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

The intrinsic functions **SQRT**, **DSQRT**, **CSQRT**, and **CDSQRT** return the square root of their respective arguments. The arguments to **SQRT** and **DSQRT** must be greater than or equal to zero. For a complex argument, **SQRT**, **CSQRT**, and **CDSQRT** return a complex number whose magnitude is the square root of the magnitude of the argument and whose angle is one-half the angle of the argument.

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 two
C     sides.
```

```
      REAL  sidea, sideb, hyp

      sidea = 3.0
      sideb = 4.0

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

      WRITE (*, 100) hyp
  100 FORMAT (/ ' The hypotenuse is ', F10.3)

      END
```

5.1.11 Exponents and Logarithms

Table 5.12 lists the intrinsic functions that return exponents or logarithms.

Table 5.12 Intrinsic Functions: Exponents and Logarithms[1]

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**

[1] Note: See Table 5.1 for a list of 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 intrinsic logarithmic functions, real arguments must be greater than zero. The **CLOG** and **CDLOG** functions return the logarithm of a complex number. The real component is the natural logarithm of the magnitude of the complex number. The imaginary component is the principal value of the angle of the complex number, in radians.

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 is assumed to be
C     proportional to the colony's size.

      REAL  sizei, sizef, time, rate

      sizei = 10000.0
      time  = 40.5
      rate  = 0.0875

      sizef = sizei * EXP (rate * time)

      WRITE (*, 100) sizef
  100 FORMAT (' THE FINAL SIZE IS ', E12.6)
      END
```

5.1.12 Trigonometric Functions

Table 5.13 summarizes the trigonometric intrinsic functions.

Table 5.13 Intrinsic Functions: Trigonometric Functions[1]

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

Table 5.13 *(continued)*

Name	Definition	Argument Type	Function Type
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**

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

All angles in trigonometric intrinsic functions are specified in radians. Table 5.14 indicates some restrictions on the arguments to and results of trigonometric intrinsic functions.

Table 5.14 Restrictions on Arguments and Results

Function	Restrictions on Arguments	Range of Results
SIN, DSIN, COS, DCOS, TAN, DTAN	None	All real numbers
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	All real numbers

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 prompts for a polar coordinate
C     and converts 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,')' )
      END
```

5.1.13 Character Functions

Table 5.15 summarizes the intrinsic functions that operate on character constants and variables.

Table 5.15 Intrinsic Functions: Character Functions[1]

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* [[, log]])	Position of substring *charB* in string *charA*	char, log	int
LEN_TRIM (*char*)	Length of string, less trailing blanks	char	int
SCAN (*char*, *charset* [[, log]])	Element of *charset* in *char*	char, log	int
VERIFY (*char*, *charset* [[, log]])	Element of *charset* not in *char*	char, log	int

[1] Note: See Table 5.1 for a list of 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 two character arguments are not of equal length, the shorter operand is padded to the length of the larger operand by adding blanks.

The argument to the **LEN** intrinsic function does not have to be assigned a value.

The **INDEX** intrinsic function returns an integer specifying 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 position of the first occurrence of *charB* is returned. The *log* parameter, when **.TRUE.**, starts the comparison at the end of the string and moves toward the beginning.

The **LEN_TRIM** function returns the length of the string argument, less the number of trailing blanks.

The **SCAN** and **VERIFY** functions both compare a string with the group of characters in *charset*. **SCAN** returns the position of the first string character that

matches a character in *charset*, while **VERIFY** returns the first position that does not match a character in *charset*. If there is no match, or the string is of zero length, **SCAN** returns zero. If there is no mismatch, or the string is of zero length, **VERIFY** returns zero. The *log* parameter, when **.TRUE.**, starts the comparison at the end of the string and moves toward the beginning.

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
`LEN_TRIM ('abc ')`	3
`INDEX ('banana', 'an', .TRUE.)`	4
`SCAN ('banana', 'nbc')`	1
`VERIFY ('banana', 'nbc')`	2

5.1.14 End-of-File Function

Table 5.16 summarizes the end-of-file intrinsic function.

Table 5.16 Intrinsic Functions: End-of-File Function[1]

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

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

If the unit specified by its argument is at or past the end-of-file record, the value **.TRUE.** is returned by the intrinsic function **EOF** (*int*). 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
C     and 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
      END
```

5.1.15 Address Functions

Table 5.17 lists the intrinsic functions that return addresses.

Table 5.17 Intrinsic Functions: Addresses[1]

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

[1] Note: See Table 5.1 for a list of 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 created 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 the **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 the **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 **$LARGE** metacommands. For example, **LOCNEAR** will not usually return the correct address of 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.

5.1.16 Bit-Manipulation Functions

Table 5.18 summarizes the functions that perform bit manipulation.

Table 5.18 Intrinsic Functions: Bit Manipulation[1]

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

Table 5.18 *(continued)*

Name	Definition	Argument Type	Function Type
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

[1] Note: See Table 5.1 for a list of abbreviations used in this table.

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

Function	Operation
Inclusive or	If the *n*th bit of either the first or second argument is 1, then the *n*th 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 *n*th bits of the first and second arguments are not equal to each other, then the *n*th bit of the result is set to 1. Otherwise, the *n*th bit of the result is set to 0.
Logical product	If the *n*th bits of both the first and second arguments are 1, then the *n*th bit of the result is set to 1. Otherwise, the *n*th bit of the result is set to 0.
Logical complement	If the *n*th bit of the argument is 1, then the *n*th bit of the result is set to 0. Otherwise, the *n*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 *intB* bit 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 5.19 shows the results of other bit-manipulation intrinsic functions.

Table 5.19 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.`

5.2 Alphabetical Function List

What follows in Table 5.20 is an alphabetical listing of all intrinsic functions in Microsoft FORTRAN. See Table 5.1 for a list of the abbreviations used for data types.

Table 5.20 Intrinsic Functions

Name	Definition	Argument Type	Function Type
ABS (*gen*)	Absolute value	int, real, or cmp	Same as argument type unless argument is cmp[1]
ACOS (*real*)	Arc cosine	real	Same as argument
AIMAG (*cmp8*)	Imaginary part of cmp8 number	cmp8	real4
AINT (*real*)	Truncate	real	Same as argument
ALLOCATED (*array*)	Allocation status of array	Any	log
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
CABS (*cmp*)	Absolute value	cmp	real[1]

Table 5.20 *(continued)*

Name	Definition	Argument Type	Function Type
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
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

Table 5.20 *(continued)*

Name	Definition	Argument Type	Function Type
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
DFLOAT (*gen*)	Data-type conversion	int, real, or cmp	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
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

Table 5.20 *(continued)*

Name	Definition	Argument Type	Function Type
DTANH (*dbl*)	Hyperbolic tangent	dbl	dbl
EOF (*int*)	End-of-file	int	log
EPSILON (*gen*)	Smallest increment over 1	real	real
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
HUGE (*gen*)	Largest positive number	int or real	Same as argument
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
IEOR (*intA*, *intB*)	Exclusive or	int	Same as argument
IFIX (*real4*)	Data-type conversion	real4	int
IMAG (*cmp*)	Imaginary part of cmp number	cmp	real[1]

Table 5.20 (*continued*)

Name	Definition	Argument Type	Function Type
INDEX (*charA*, *charB* [[, *log*]])	Location of substring *charB* in string *charA*	char, log	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
LEN_TRIM (*char*)	Length of string, excluding trailing blanks	char	int
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

Table 5.20 *(continued)*

Name	Definition	Argument Type	Function Type
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
MAXEXPONENT (*real*)	Largest positive exponent of data type	real	real
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
MINEXPONENT (*real*)	Largest negative exponent of data type	real	real
MOD (*genA*, *genB*)	Remainder	int or real	Same as argument
NEAREST (*real*, *director*)	Nearest value in direction of sign of director	real	real
NINT (*real*)	Round	real	int
NOT (*intA*)	Logical complement	int	Same as argument
PRECISION (*gen*)	Number of significant digits for data type	real	int
REAL (*gen*)	Data-type conversion	int, real, or cmp	real4

Table 5.20 *(continued)*

Name	Definition	Argument Type	Function Type
SCAN (*charA*, *charB* [[,*log*]])	Position of first occurrence of character from *charB* in *charA*	char and log	int
SIGN (*genA*, *genB*)	Sign transfer	int or real	Same as argument
SIN (*gen*)	Sine	real or cmp	Same as argument
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
TINY (*real*)	Returns smallest positive number > 0 for data type	real	real
VERIFY (*charA*, *charB* [[,*log*]])	Position of first occurrence of character not from *charB* in *charA*	char and log	int

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

5.3 Additional Procedures

Microsoft FORTRAN contains additional procedures that control and access system time and date, get and reset run-time error-code information, return command-line arguments, and generate pseudorandom numbers. The following sections describe these procedures.

These procedures are included in the FORTRAN run-time library, and are automatically linked to your program if called. However, they are not intrinsic

functions. You may write other functions that appropriate these functions' names without having to reference the names in an **EXTERNAL** statement. Your own functions will be linked instead, as long as LINK references their object code before calling the FORTRAN library.

NOTE *Microsoft FORTRAN Advanced Topics discusses other functions in the FORTRAN libraries, including the C spawnlp and system functions (Chapter 3), graphics and full-screen text functions (Chapters 10–13), and OS/2 thread control functions (Chapter 8).*

5.3.1 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 5.21 summarizes the time and date procedures.

Table 5.21 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, array elements, or structure 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 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.

Example

The following program sets the date and time, then prints them on the screen:

```
C     Warning: this program will reset
C     your 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 (2001, 7, 4)))
     +     WRITE (*, *) 'SETDAT failed'
C     sets the date to July 4th, 2001:
      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
      WRITE (*, '(1X, A, I2.2, 1H/, I2.2, 1H/, I4.4)')
     +cdate, imon, iday, iyr
C     writes the date
      WRITE (*, '(1X, A, I2.2, 1H:, I2.2, 1H:, I2.2, 1H.,
     +I2.2)') ctime, ihr, imin, isec, i100th
C     writes the time in the format xx:xx:xx.xx
      END
```

5.3.2 Run-Time-Error Procedures

The **IGETER** function and the **ICLRER** subroutine are included for compatibility with previous versions of FORTRAN. Their functionality is provided in the current version by the **IOSTAT=** option. (See Section 3.2.6, "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. The significance of the value returned is explained in the following list:

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 takes the following form:

```
SUBROUTINE ICLRER (iunit)
INTEGER*2 iunit
.
.
.
END
```

5.3.3 Command-Line-Argument Procedures

The **NARGS** function returns the total number of command-line arguments, including the command. The **GETARG** subroutine returns the *n*th command-line argument (where the command itself is argument number zero). The syntax of these procedures is shown below:

numargs = **NARGS ( )**
CALL GETARG (*n***,** *buffer***,** *status***)**

The **NARGS** function takes no arguments. It always returns an **INTEGER*4** value, regardless of the **$STORAGE** setting.

The **GETARG** subroutine takes three arguments. The first, of **INTEGER*2** type, specifies the position of the desired argument. (The command itself is argument zero.)

The *buffer* argument is a **CHARACTER** variable that returns the desired command-line argument. If the argument is shorter than *buffer*, **GETARG** pads *buffer* on the right with blanks. If the argument is longer than *buffer*, **GETARG** truncates the argument.

The *status* argument is an **INTEGER*2** type that returns a status value on completion. If there were no errors, *status* returns the number of characters in the command-line argument before truncation or blank-padding. (That is, *status* is the original number of characters in the command-line argument.) Errors include

specifying an argument position less than zero or greater than the value returned by **NARGS**. For either of these errors, *status* returns a value of –1.

The *status* argument is an **INTEGER*2** type which returns a *status* value.

If there were no errors, *status* contains the number of characters in the command-line argument before truncation or blank-padding. (That is, *status* is the original number of characters in the command-line argument.) Errors include passing a *status* less than one, or specifying an argument position less than zero or greater than the value returned by **NARGS**. For any of these errors, *status* returns a value of –1.

Example

Assume a command-line invocation of ANOVA -g -c -a, and that *buffer* is at least five characters long. The following **GETARG** statements return the corresponding arguments in the buffer:

Statement	String Returned	Length Returned
`CALL GETARG (0, buffer, status)`	`ANOVA`	5
`CALL GETARG (1, buffer, status)`	`-g`	2
`CALL GETARG (2, buffer, status)`	`-c`	2
`CALL GETARG (3, buffer, status)`	`-a`	2
`CALL GETARG (4, buffer, status)`	`undefined`	-1

5.3.4 Random Number Procedures

The **RANDOM** subroutine returns a pseudorandom real value greater than or equal to zero and less than one. The **SEED** subroutine changes the starting point of the pseudorandom number generator. The syntax of these procedures is shown below:

CALL RANDOM (*ranval*)
CALL SEED (*seedval*)

The **RANDOM** subroutine takes a single **REAL*4** argument through which the random value is returned.

The **SEED** subroutine takes a single **INTEGER*2** argument. **SEED** uses this value to establish the starting point of the pseudorandom number generator. A given seed always produces the same sequence of values from **RANDOM**.

If **SEED** is not called before the first call to **RANDOM**, **RANDOM** always begins with a seed value of one. If a program must have a different pseudorandom sequence each time it runs, use the **GETTIM** procedure to pick a seed value (the *hundredth-of-a-second* parameter is a good choice because it changes so rapidly).

5.3.5 Executing DOS System Calls

The **INTDOSQQ** and **INTDOSXQQ** subroutines invoke DOS system calls.

INTDOSXQQ can specify a segment register value, making it suitable for programs with large-model data segments or far pointers that need to specify which segment to use during the system call. **INTDOSQQ** does not specify a segment register value, making it suitable for programs with single data segments or near pointers.

The syntax for the routines are similar:

CALL INTDOSQQ (*inregs*, *outregs*)

CALL INTDOSXQQ (*inregs*, *outregs*, *segregs*)

Argument	Description
inregs	Structure containing register values on call
outregs	Structure containing register values on return
segregs	Structure containing regment register values on call

The **INTERFACE** and declarations for these routines and structures are provided in FLIB.FI and FLIB.FD, respectively.

To invoke a system call, both functions execute an INT 21H instruction. Before executing the instruction, the functions copy the contents of *inregs* (and *segregs* for **INTDOSXQQ**) to the corresponding registers. For **INTDOSXQQ**, only the DS and ES register values in *segregs* are used.

After the INT instruction returns, the functions copy the current register values to *outregs*. A nonzero return value in the **cflag** field of the **outregs** structure indicates an error condition.

INTDOSQQ is used to invoke DOS system calls that do not use segment registers. **INTDOSXQQ** is used to invoke DOS system calls that take an argument in the ES register or that take a DS register value different from the default data segment.

5.3.6 Signal Handling

The **SIGNALQQ** and **RAISEQQ** functions control interrupt signal handling. **SIGNALQQ** identifies the source of an interrupt signal (so a routine can respond appropriately), while **RAISEQQ** has the ability to generate several types of interrupt signals.

SIGNALQQ lets a routine determine how to respond to an interrupt signal from the operating system. **SIGNALQQ** uses the following syntax:

iret = **SIGNALQQ** (*sig*,*func*)

Variable	Description
iret	Integer return value
sig	Signal value
func	Function to be executed

The *sig* argument must be one of the following constants:

Value	Modes	Meaning	Default Action
SIG$ABRT	Real, protected	Abnormal termination	Terminates the calling program with exit code 3
SIG$BREAK	Protected	CTRL+BREAK signal	Terminates the calling program with exit code 3
SIG$FPE	Real, protected	Floating-point error	Terminates the calling program with exit code 3
SIG$ILL	Real, protected	Illegal instruction	Terminates the calling program with exit code 3
SIG$INT	Real, protected	CTRL+C signal	Terminates the calling program with exit code 3
SIG$SEGV	Real, protected	Illegal storage access	Terminates the calling program with exit code 3
SIG$TERM	Real, protected	Termination request	Terminates the calling program with exit code 3
SIG$USR1	Protected	OS/2 process flag A	Signal is ignored
SIG$USR2	Protected	OS/2 process flag B	Signal is ignored
SIG$USR3	Protected	OS/2 process flag C	Signal is ignored

SIG$USR1, **SIG$USR2**, and **SIG$USR3** are user-defined signals which can be sent by means of **DosFlagProcess.** For details, see *Microsoft Operating System/2 Programmer's Reference.*

Note that **SIG$ILL**, **SIG$SEGV**, and **SIG$TERM** are not generated under DOS and **SIG$SEGV** is not generated under OS/2. They are included for ANSI C compatibility. Thus, you may set signal handlers for these signals via **SIGNALQQ**, and you may also explicitly generate these signals by calling **RAISEQQ.**

The *func* argument is a function address. When **SIGNALQQ** recieves a signal of the type specified by *sig*, it installs the function *func* as the handler for that signal.

NOTE *All signal handler functions need to be declared with the C attribute.*

For all signals except **SIG$FPE** and **SIG$USR***X*, the function is passed the *sig* argument **SIG$INT** and executed.

For **SIG$FPE** signals, the function is passed two arguments; namely **SIG$FPE** and the floating-point error code identifying the type of exception that occurred.

For **SIG$USR1**, **SIG$USR2**, and **SIG$USR3**, the function is passed two arguments: the signal number and the argument furnished by the **DosFlagProcess** function.

For **SIG$FPE**, *func* is passed two arguments, **SIG$FPE** and an integer error subcode, **FPE$***xxx*; then the function is executed. (See the include file FLIB.FD for definitions of the **FPE$***xxx* subcodes.)

If *func* returns, the calling process resumes execution immediately following the point at which it received the interrupt signal. This is true regardless of the type of signal or operating mode.

Since signal-handler routines are normally called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a run-time operation is incomplete and in an unknown state. Certain restrictions therefore apply to the routines that can be used in your signal-handler routine:

1. Do not issue low-level or standard input and output routines (e.g., **READ** and **WRITE**).
2. Do not call heap routines or any routine that uses the heap routines (e.g., I/O, **ALLOCATE**, **DEALLOCATE**).
3. Do not use any overlay routines.

A return value of **SIG_ERR** indicates an error.

The **RAISEQQ** function sends an interrupt signal to the executing program, simulating an interrrupt signal from the operating system.

RAISEQQ has the following syntax:

iret = **RAISEQQ** (*sig*)

Variable	Description
iret	Integer return value
sig	Signal to raise

If a signal-handling routine for *sig* has been installed by a prior call to **SIGNALQQ**, **RAISEQQ** causes that routine to be executed. If no handler routine has been installed, the default action (as listed below) is taken.

The signal value *sig* can be one of the following constants:

Signal	Meaning	Default
SIG$ABRT	Abnormal termination.	Terminates the calling program with exit code 3.
SIG$BREAK	CTRL+ BREAK interrupt.	Terminates the calling program with exit code 3.
SIG$FPE	Floating-point error.	Terminates the calling program.
SIG$ILL	Illegal instruction. This signal is not generated by DOS or OS/2, but is supported for ANSI C compatibility.	Terminates the calling program.
SIG$INT	CTRL+ C interrupt.	Issues INT23H.
SIG$SEGV	Illegal storage access. This signal is not generated by DOS or OS/2, but is supported for ANSI C compatiblity.	Terminates the calling program.
SIG$TERM	Termination request sent to the program. This signal is not generated by DOS or OS/2, but is supported for ANSI C compatibility.	Ignores the signal.
SIG$USR1 **SIG$USR2** **SIG$USR3**	User-defined signals.	Ignores the signal.

If **RAISEQQ** is successful, it returns a value of 0. Otherwise, it returns a nonzero value.

5.3.7 Handling Math Errors

Whenever a math function generates an error, the FORTRAN run-time library automatically calls the subroutine **MATHERRQQ**. If **MATHERRQQ** does not resolve the error, the process is halted with an appropriate run-time error. The definition of **MATHERRQQ** supplied with the run-time library handles the error by stopping the process. If you wish to proceed, you'll need to write your own **MATHERRQQ** definition.

The declaration of **MATHERRQQ** must correspond to the following syntax:

SUBROUTINE MATHERRQQ (*name*, *nlen*, *info*, *retcode*)

Argument	Description
name	A character array that receives the name of the function causing the error
nlen	The length of the *name* array
info	A record containing the data types of the values causing the error
retcode	A return code passed back to the run-time indicating whether the error was successfully resolved

The *name* parameter is a **CHARACTER** array which the run-time fills with the name of the function causing the error. The run-time fills the second parameter, the integer *nlen*, with the length of the *name* array.

The *info* record contains elements that pass **MATHERRQQ** the following information:

- A code indicating the type of error that occurred (see below)
- The data type of the function that caused the error
- The argument(s) that were passed to the function and generated the error
- The correct value to return to the host program from the function that caused the error

The elements of the *info* record are explicitly defined in FLIB.FD. The **ERRCODE** element specifies the type of math error that occurred, and can have one of the following values:

Value	Meaning
MTH$E_DOMAIN	Argument domain error
MTH$E_SINGULARITY	Argument singularity
MTH$E_OVERFLOW	Overflow range error
MTH$E_PLOSS	Partial loss of significance
MTH$E_TLOSS	Total loss of significance
MTH$E_UNDERFLOW	Underflow range error

The **FTYPE** element of the *info* structure identifies the data type of the math function. The third element is a **UNION** containing the function argument(s) and the value that you want the math function to return to the program. The value you indicate usually depends on the information in the other *info* structure elements.

If the error is not recoverable, set the return code *retcode* to 0 to indicate an error. Alternately, set it to a nonzero value to indicate successful corrective action.

The following short program illustrates the general process of redefining **MATHERRQQ**.

```
INCLUDE 'FLIB.FI'

PROGRAM MAIN

INCLUDE 'FLIB.FD'

PRINT *, LOG (-1.0)
END

SUBROUTINE MATHERRQQ (NAME, NLEN, INFO, RETCODE)
INCLUDE 'FLIB.FD'
INTEGER*2 NLEN
CHARACTER NAME (NLEN)
RECORD /MTH$E_INFO/ INFO
INTEGER*2 RETCODE

PRINT *, NLEN
PRINT *, NAME
PRINT *, INFO.ERRCODE
PRINT *, INFO.FTYPE
INFO.R8RES = 0.0
RETCODE = 2
RETURN
END
```

PART 2

Compiling and Linking

PART 2

Compiling and Linking

The two chapters in this section provide information on using metacommands for compiler control and on using the FL command for compiling and linking.

Chapter 6, which describes metacommands, includes overview material as well as an alphabetical metacommand directory. Chapter 7 shows how to use FL to compile and link FORTRAN programs in a single step. This chapter includes detailed information on available FL options.

CHAPTERS

CHAPTER 6

Metacommands

6

The first part of this chapter lists all Microsoft FORTRAN metacommands. The second part explains how the conditional-compilation metacommands are used to control which sections of your program are compiled. The third part is a directory of metacommands, arranged alphabetically.

Metacommands tell the Microsoft FORTRAN Compiler how you want the source code compiled. Note that some command-line options, as described in *Microsoft FORTRAN Advanced Topics*, duplicate metacommand functions. If there is a conflict between a metacommand and a command-line switch, the metacommand takes precedence.

Table 6.1 summarizes the Microsoft FORTRAN metacommands. Although some metacommands (such as **$DO66** or **$FREEFORM**) can be used only once in a program, many can appear anywhere in the source code. This flexibility permits specific compilation features (such as loop optimization) to be enabled or disabled as desired.

Table 6.1 Metacommands

Metacommand	Instructions to Compiler	Default
$DEBUG[[*:string*]]	Turns on run-time checking for integer arithmetic operations, assigned **GOTO** values, subscript bounds, substrings, and **CASE** selection. **$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**

Table 6.1 *(continued)*

Metacommand	Instructions to Compiler	Default
$DEFINE *symbol-name* [[= *val*]]	Creates (and optionally assigns a value to) a variable whose existence may be tested during conditional compilation. **$UNDEFINE** removes a symbolic variable name.	**$UNDEFINE**
$DO66	Uses FORTRAN 66 semantics for **DO** statements.	**$DO66** not set
$ELSE	Marks the beginning of a conditional compilation block that is compiled if the logical condition in the matching **$IF** metacommand is **.FALSE.**.	None
$ELSEIF *expr*	Marks the beginning of a new conditional compilation block that is compiled if the logical condition in the matching **$IF** metacommand is **.FALSE.** and *expr* is **.TRUE.**.	None
$ENDIF	Terminates **$IF**, **$IF...$ELSE**, or **$IF...$ELSEIF** conditional compilation blocks.	None
$FLOATCALLS	Generates calls to subroutines in the emulator library. **$NOFLOATCALLS** causes the compiler to generate in-line interrupt instructions.	**$NOFLOATCALLS**
$FREEFORM	Uses free-form format for source code. **$NOFREEFORM** uses fixed format.	**$NOFREEFORM**
$IF *expr*	Marks the beginning of a conditional compilation block. The succeeding statements are compiled if the conditional expression is **.TRUE.**.	None
$INCLUDE:'*file*'	Proceeds as if the contents of *file* 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**
$LIST	Begins generation of listing information that is sent to the listing file. **$NOLIST** suppresses generation of listing information.	**$LIST**
$LOOPOPT	**$LOOPOPT** optimizes loops in following code. **$NOLOOPOPT** disables loop optimization.	**$LOOPOPT**

Table 6.1 *(continued)*

Metacommand	Instructions to Compiler	Default
$MESSAGE:*string*	Sends a character string to the standard output device.	None
$PACK:*n*	Sets number of bytes for packing width. *n* must be 1, 2, or 4.	**$PACK:2**
$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** [1]
$TITLE:*title*	Uses title *title* for subsequent pages of listing.	**$TITLE:"C** [1]
$TRUNCATE	Truncates variables to six characters. **$NOTRUNCATE** turns off truncation.	**$NOTRUNCATE**

[1] A null C string; that is, there is no title or subtitle.

Any line with a dollar sign ($) 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.

6.1 Using Conditional-Compilation Metacommands

In FORTRAN, the conditional-execution statements **IF...THEN...ELSE**, **ELSE**, and **ELSE IF** control the execution of statement blocks. In a similar fashion, the conditional-compilation metacommands control which sections of source code are compiled. These special metacommands make it easy to include or omit test code, customize code for specific applications by controlling which sections are included, or bypass incomplete code during development.

Like all metacommands, conditional-compilation metacommands begin with a dollar sign and must start in column one. Otherwise, they look and work like their FORTRAN counterparts:

$IF *cond-expr*
$ELSE
$ELSEIF *cond-expr*
$ENDIF

In conditional-compilation metacommands, conditional expressions (*cond-exprs*) can take several forms. The simplest is a FORTRAN logical value, **.TRUE.** or **.FALSE.**.

```
$IF .TRUE.
      WRITE (*, '('' This is compiled if .TRUE.'')')
$ELSE
      WRITE (*, '('' This is compiled if .FALSE.'')')
$ENDIF
```

In the above example, the first block of code is compiled and the second is ignored. If **.TRUE.** were changed to **.FALSE.**, the second block would be the only block compiled. However, the programmer must manually locate every **.TRUE.** or **.FALSE.** to be changed. An alternative is to create one or more symbolic names at the beginning of the program, using the **$DEFINE** metacommand:

$DEFINE *symbol-name*

The presence or absence of a particular definition is tested by using the **DEFINED** (*symbol*) conditional expression in the **$IF** metacommand. **DEFINED** (*symbol*) is **.TRUE.** if *symbol* appeared in a previous **$DEFINE** metacommand, without having been assigned a value. The advantage of this arrangement is the way the programmer can alter the meaning of a symbol throughout the program simply by adding or deleting a single metacommand. It is simpler and more reliable than changing every **.TRUE.** to **.FALSE.**.

For example, once the `$DEFINE logicvar` metacommand appears in a program, any subsequent `$IF DEFINED (logicvar)` metacommand evaluates to **.TRUE.**. If `$DEFINE logicvar` has not appeared, `$IF DEFINED (logicvar)` evaluates to **.FALSE.** as in the following examples:

```
$DEFINE truthvar

$IF DEFINED (truthvar)
      WRITE (*, '('' This is compiled if truthvar     DEFINED'')')
$ELSE
      WRITE (*, '('' This is compiled if truthvar not DEFINED'')')
$ENDIF
```

Since the symbol `truthvar` appears in `$DEFINE truthvar`, `DEFINED (truthvar)` evaluates to **.TRUE.** and the first block of statements is compiled. If the `$DEFINE truthvar` metacommand were removed (or converted to a comment line), `truthvar` would not be defined, `DEFINED`

(truthvar) would evaluate to **.FALSE.**, and the second block of statements would be compiled.

The **$UNDEFINE** metacommand cancels a symbol definition. In the following example, the **$UNDEFINE** metacommand cancels truthvar, and DEFINED truthvar evaluates to **.FALSE.**. The compiler ignores the first statement block:

```
$DEFINE truthvar
$UNDEFINE truthvar

$IF DEFINED (truthvar)
      WRITE (*, '('' This is compiled if truthvar     DEFINED'')')
$ELSE
      WRITE (*, '('' This is compiled if truthvar not DEFINED'')')
$ENDIF
```

You may define as many symbolic names as you wish anywhere in the program. These symbols will not conflict with FORTRAN identifiers or variable names. You may use names like data, sin, equivalence, or the name of an external procedure, if you wish.

A **$DEFINE** metacommand can also give an integer value to a symbol name (any four-byte FORTRAN integer value is allowed):

```
$DEFINE choice = 100000

$IF choice .NE. 100000
      WRITE (*, '('' This is compiled if choice .NE. 100000'')')
$ELSE
      WRITE (*, '('' This is compiled if choice .EQ. 100000'')')
$ENDIF
```

In the example above, choice .NE. 100000 evaluates to **.FALSE.**, so the second statement block is compiled. (DEFINED (choice) would also evaluate to **.FALSE.**, because choice was assigned a value.) This example also shows how a conditional expression can include any FORTRAN relational operator (**.EQ.**, **.NE.**, **.LT.**, **.LE.**, **.GT.**, or **.GE.**). The FORTRAN **.NOT.** operator can also be used:

```
$DEFINE choice = 100000
$DEFINE receiver = choice

$IF .NOT. (receiver .NE. 100000)
      WRITE (*, '('' This is compiled if receiver .EQ. 100000'')')
$ELSE
      WRITE (*, '('' This is compiled if receiver .NE. 100000'')')
$ENDIF
```

In the example above, the second **$DEFINE** metacommand gives the value of choice to receiver. (This assignment is not permitted unless choice has already been given an integer value.) The **.NOT.** operator reverses the logical value of receiver .NE. 100000, and the first statement block is compiled.

The ability to assign integer variables to symbol names greatly increases the programmer's control over whether a statement block is compiled, as shown below:

```
$DEFINE upper = 3
$DEFINE lower = -1

$IF upper .LT. 3
      WRITE (*, '('' Compiled if upper .LT. 3'')')
$ELSEIF lower .GT. -2
      WRITE (*, '('' Compiled if upper .GE. 3 and lower .GT. -2'')')
$ENDIF
```

The FORTRAN conditional operators **.AND.** and **.OR.** can also be used in comparisons:

```
$DEFINE upper
$DEFINE lower = -1

$IF (DEFINED (upper) .OR. (lower .LE. -2)) .AND. middle
      WRITE (*, '('' Compiled if all conditions met'')')
$ENDIF
```

In this example, `upper` is defined, but `lower` is not less than or equal to –2. However, the **.OR.** comparison is enclosed within parentheses, so the net evaluation of these two statements is **.TRUE.**. The symbol `middle` was not defined, so `middle` is **.FALSE.**, making the full expression **.FALSE.** as well. The enclosed statement block is not compiled.

Logical expressions may be as complex as desired, and may include parentheses wherever needed to clarify the logic or override precedence. (The rules of precedence and logical association are the same as in FORTRAN.) However, the expression must fit on one line.

Comments may be added at the end of metacommand lines. Comments must begin with an exclamation point:

```
$DEFINE test          ! controls compilation of test code
```

Constantly editing a file to add or delete symbol declarations can be inconvenient. Therefore, you are also permitted to define symbols on the compiler command line using the /D command-line option. The option /D*symbol* defines `symbol` at the beginning of the compilation. (A subsequent **$UNDEFINE** metacommand within the program file can cancel the definition.) The option /D*symbol=integer* defines the symbol and gives it an integer value.

Each /D option can define or assign only one symbol. However, several /D options can be included in one command line. The only real limit is the maximum length of a command line set by your operating system.

6.2 Metacommand Directory

The remainder of this chapter is an alphabetical directory of the 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 fragments that illustrate the use of the metacommand. This section does not appear with every reference entry.

ACTION **$DEBUG** directs the compiler to perform additional testing and expanded error handling, and can also be used for conditional compilation; **$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, enabling and disabling the debug features as desired.

When **$DEBUG** is enabled, 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 run-time errors occur, the file name and line number are displayed on the console.
- Checks range of subscripts and 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 **$DEBUG** is not enabled, the variable is truncated, no error is reported, and the program returns unpredictable results. In the case of real numbers, an error is always reported.
- Checks for a **CASE DEFAULT** statement. If there is no **CASE DEFAULT** in a **SELECT CASE** statement, and the value of the test expression does not match any value in any **CASE** expression list, program execution halts with an error message. (If you do not use **$DEBUG**, no error occurs, and execution passes beyond the **SELECT CASE** construct to the next executable statement.)

NOTE ***$DEBUG*** *has no effect on floating-point exception handling. See Microsoft FORTRAN Advanced Topics for information on exception handling on your system.*

This metacommand should be placed in each source file to be compiled.

If the optional *string* is specified, the characters in *string* specify that lines with 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 **$DEBUG:***string* metacommand is specified, each *string* overrides the previous *string*. **$DEBUG** can be used for conditional compilation only if the **$FREEFORM** metacommand has not been specified. If the **$DEBUG:***string* metacommand appears after **$FREEFORM**, the compiler emits a warning message.

EXAMPLE

```
C     If the $FREEFORM metacommand has been specified,
C     the 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.
```

ACTION **$DECLARE** generates warnings for variables that have not appeared in specification statements, and **$NODECLARE** disables these warnings

SYNTAX **$[[NO]]DECLARE**

REMARKS The default is **$NODECLARE**. When **$DECLARE** is enabled, a warning message is generated at the first use of any variable that has not been assigned a value in a specification statement. The **$DECLARE** metacommand is primarily a debugging tool that locates variables that have not been properly initialized, or that have been defined but never used.

EXAMPLE

```
$DECLARE
C     Since the variable z has not been assigned a value,
C     its use in the statement labeled 100 causes an error:

      REAL x, y, z
      y = 1.0
100   x = y + z
```

ACTION

$DEFINE creates a symbolic variable whose existence or value can be tested during conditional compilation; **$UNDEFINE** removes the symbol

SYNTAX

$DEFINE *symbol-name* [[= *val*]]
$UNDEFINE *symbol-name*

Parameter	Description
symbol-name	An alphanumeric identifier of up to 31 characters. It may include the dollar sign and underline; it may not begin with a numeral or the underline.
val	Any positive or negative **INTEGER*4** value.

REMARKS

The default is **$UNDEFINE**, since any symbol name not specified in a **$DEFINE** metacommand tests **.FALSE.**.

The existence of a *symbol-name* is tested by the **DEFINED** (*symbol-name*) conditional expression. The value of a *symbol-name* is tested with a FORTRAN-style conditional expression. If a *symbol-name* has been assigned a value, it tests **.FALSE.** in a **DEFINED** conditional expression.

Symbol names are local to the metacommands, and may duplicate FORTRAN keywords, intrinsic functions, or user-defined names, without conflict.

The **$DEFINE** and **$UNDEFINE** metacommands can appear anywhere in a program, enabling symbol definitions as desired.

EXAMPLES

```
$DEFINE  testflag
$DEFINE  testval = 2
.
.
.
$UNDEFINE  testflag
```

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 enabled, the following FORTRAN 66 semantics are used:

- Statements within a **DO** loop are always executed at least once.
- Extended range is permitted; 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 will not be executed 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; 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, however.)

ACTION

Marks the beginning of a **$ELSE** metacommand block

SYNTAX

$ELSE

REMARKS

If the logical expression in the matching **$IF** or **$ELSEIF** metacommand evaluates to **.FALSE.**, a **$ELSE** block is compiled.

A **$ELSE** block consists of any statements between the **$ELSE** metacommand and the next **$ENDIF** metacommand at the same **$IF** level. The matching **$ENDIF** must appear before any **$ELSE** or **$ELSEIF** at the same **$IF** level.

EXAMPLE

```
$DEFINE flag

$IF DEFINED (flag)
      WRITE (*, '('' This is compiled if flag
     +               DEFINED'')')
$ELSE
      WRITE (*, '('' This is compiled if flag
     +               not DEFINED'')')
$ENDIF
```

ACTION

Causes compilation of a block of statements if *expression* is **.TRUE.** and the matching **$IF** metacommand is **.FALSE.**

SYNTAX

$ELSEIF (*expression*)

Parameter	Description
expression	A logical expression

REMARKS

The logical expression in a **$ELSEIF** metacommand can take two forms. One is the conditional-compilation operator **DEFINED**, followed by a symbol name in parentheses. If the symbol name has previously appeared in a **$DEFINE** metacommand without having been assigned a value, **DEFINED** (*symbol_name*) evaluates to **.TRUE.**. If (*symbol_name*) did not appear in a preceding **$DEFINE** metacommand, or did appear and was assigned a value, **DEFINED** (*symbol_name*) evaluates to **.FALSE.**.

The second form is a logical comparison, where the value of the symbol is compared with an integer constant or the value of another symbol using the FORTRAN **.EQ.**, **.NE.**, **.GT.**, **.LT.**, **.GE.**, or **.LE.** operators. The results of such comparisons may further be evaluated with the FORTRAN **.AND.**, **.OR.**, **.XOR.**, and **.NOT.** operators. The usual rules of precedence apply, and parentheses may be used to control the order of evaluation.

The associated **$ELSEIF** block consists of any executable statements between the **$ELSEIF** metacommand and the next **$ELSEIF**, **$ELSE**, or **$ENDIF** metacommand at the same **$IF** level.

EXAMPLE

```
$DEFINE flag
$DEFINE testval = 3

$IF (.NOT. DEFINED (flag))
      WRITE (*, '('' Compiled if flag not DEFINED'')')
$ELSEIF (testval .GT. 3)
      WRITE (*, '('' Compiled if flag DEFINED .AND. testval
     +.GT. 3'')')
$ELSE
      WRITE (*, '('' Compiled if flag DEFINED .AND. testval
     +.LE. 3'')')
$ENDIF
$ENDIF
```

ACTION Terminates a **$IF** metacommand block

SYNTAX **$ENDIF**

REMARKS There must be a matching **$ENDIF** metacommand for every **$IF** metacommand in a program unit.

EXAMPLE

```
$DEFINE  flag
$DEFINE  testval = 3
$IF DEFINED (flag)
      WRITE (*, '('' Compiled if all conditions met'')')
$ENDIF
.
.
.
$IF (testval .LT. 1) .AND. .NOT. DEFINED (flag)
      WRITE (*, '('' This is compiled if flag .EQ. 3'')')
$ELSE
      WRITE (*, '('' This is compiled if flag .NE. 3'')')
$ENDIF
```

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 *Microsoft FORTRAN Advanced Topics* for a discussion of the advantages and disadvantages of floating-point operations for handling 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

C     The function calculates the sine of X using a power series.
C     Successive terms are calculated until less than eps.

C     Library calls are generated instead of in-line instructions,
C     letting this routine run on machines without a coprocessor.

      REAL FUNCTION SINE (x, eps)

      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
      END IF
      SINE = y
      END
```

ACTION

$NOFREEFORM specifies that a source file is in standard FORTRAN format; **$FREEFORM** specifies that a source file is in free-form format

SYNTAX

$[[NO]]FREEFORM

REMARKS

If this metacommand appears, it must precede any FORTRAN statements. The default, **$NOFREEFORM**, tells the compiler that your source code is in the standard FORTRAN format: labels are in columns 1–5, continuation characters are in column 6, statements are in columns 7–72, and characters beyond column 73 are ignored. The standard FORTRAN format is described in Section 2.1, "Lines." **$FREEFORM** tells the compiler that your source code is in the following format:

- A double quotation mark (") in column 1 indicates a comment line. An exclamation point outside a character or Hollerith constant indicates the beginning of an in-line comment.
- 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, noncomment 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 number x is calculated using a power series.
"Successive terms are calculated until one is less than epsi.

REAL x, epsi, z, sine, next
epsi = 0.0001

      WRITE  (*, 100)
 100  FORMAT (1X, 'ENTER x: ' \)
      READ   (*, '(F10.5)') x

      z    =  AMOD (x, 6.2831853)
      sine =  z
      i    =  4.0
      next = -z * z * z / 6.0
```

```
200   IF (ABS (next) .GE. epsi) THEN
         sine =  sine + next
         next = -next * z * z / (i * (i + 1.0))
         i    =  i + 2.0
         GOTO 200
      ENDIF

      WRITE  (*, 300) x, sine
300   FORMAT (1X, 'THE SINE OF ', F10.5, -
            ' = ', F12.10)
      END
```

ACTION

If *expression* is **.TRUE.**, statements in the **$IF** block are compiled; if *expression* is **.FALSE.**, control is transferred to the next **$ELSE**, **$ELSEIF**, or **$ENDIF** metacommand at the same **$IF** level

SYNTAX

$IF *expression*

Parameter	Description
expression	A logical expression

REMARKS

The logical expression in a **$IF** metacommand can take two forms. One is the metacommand keyword **DEFINED**, followed by a symbol name in parentheses. If the symbol name has previously appeared in a **$DEFINE** metacommand without having been assigned a value, **DEFINED** (*symbol_name*) evaluates to **.TRUE.**. If (*symbol_name*) did not appear in a preceding **$DEFINE** metacommand, or did appear and was assigned a value, **DEFINED** (*symbol_name*) evaluates to **.FALSE.**.

The second form is a logical comparison, where the value of the symbol is compared with an integer constant or the value of another symbol using the FORTRAN **.EQ.**, **.NE.**, **.GT.**, **.LT.**, **.GE.**, or **.LE.** operators. The results of such comparisons may further be evaluated with the FORTRAN **.AND.**, **.OR.**, **.XOR.**, and **.NOT.** operators. The usual rules of precedence apply, and parentheses may be used to control the order of evaluation.

EXAMPLE

```
$DEFINE flag
$DEFINE testval = 3
$IF DEFINED (flag)
      WRITE (*, '('' This is compiled if flag DEFINED'')')
$ENDIF
.
.
.
$IF (testval .LT. 1) .AND. .NOT. DEFINED (flag)
      WRITE (*, '('' Testval .LT. 1 .AND. flag .NOT. DEFINED'')')


$ELSE
      WRITE (*, '('' Testval .GE. 1 .OR.  flag       DEFINED'')')
$ENDIF
```

ACTION Inserts the contents of a specified text file at the location of the **$INCLUDE** metacommand

SYNTAX **$INCLUDE:**'*filename*'

Parameter	Description
filename	Name of the FORTRAN text file to include in the program

REMARKS The argument *filename* must be a valid file specification for your operating system.

The compiler considers the contents of the include file to be part of the program file and compiles them immediately. At the end of the included file, the compiler resumes processing the original source file at the line following the **$INCLUDE** metacommand.

Include files are primarily used for data or program units that appear in more than one program. Include files most often contain subroutines and functions, common block declarations, and **EXTERNAL**, **INTERFACE TO**, and **INTRINSIC** statements.

Include files can also contain other **$INCLUDE** metacommands and **INCLUDE** statements (nested included files). The compiler allows you to nest any combination of up to ten **$INCLUDE** metacommands or **INCLUDE** statements. Your operating system may impose further restrictions.

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 inserted in the source code in place of every **$INCLUDE** metacommand. This guarantees 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 out the numbers in reverse order:
      WRITE (*, *) ' '
      DO 200 i = 1,5
         CALL Pop (x)
         WRITE (*, *) x
  200 CONTINUE
      END
```

```
      SUBROUTINE Push (x)
C     Pushes an element x onto the top of the stack.

      REAL x
$INCLUDE:'stkvars.for'

      top = top + 1
      IF (top .GT. stacksize)  STOP 'Stack overflow'
      stack(top) = x
      END

      SUBROUTINE pop(x)
C     Pops an element from the top of the stack into x.

      REAL x
$INCLUDE:'stkvars.for'

      IF (top .LE. 0)  STOP 'Stack underflow'
         x   = stack(top)
         top = top - 1
      END
```

The following is the file STKVARS.FOR:

```
C     This file contains the declaration of the common block
C     for a stack implementation. Because this file contains an
C     assignment statement, it must be included only after all
C     other specification statements in each program unit.

      REAL     stack(500)
      INTEGER  top, stacksize

      COMMON  /stackbl/ stack, top

      stacksize = 500
```

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

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 **$LARGE** metacommand, it indicates that the array or formal array argument specified can span more than one segment (it is addressed outside of DGROUP). When *names* is specified in the **$NOTLARGE** metacommand, it excludes the specified items from the effects of a **$LARGE** metacommand that has no arguments.

REMARKS

$NOTLARGE is the default.

If the optional *names* parameter is specified, the metacommand must appear 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 without arguments may appear anywhere except in the executable section of a subprogram.

Arrays with explicit dimensions indicating they are bigger than 64K are automatically allocated to multiple segments outside 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
      .
      .
      .
```

You can also use the **HUGE** attribute to specify that an actual argument can span more than one segment.

ACTION **$LINESIZE** formats subsequent pages of the listing to a width of *n* columns

SYNTAX **$LINESIZE:***n*

Parameter	Value
n	An integer between 80 and 132. The default for *n* is 80.

EXAMPLE

```
$LINESIZE:100

C     The compiler listing of this program will be one hundred
C     columns wide:

      REAL x
      x = 20
      WRITE (*, 100) x, SQRT(x)
  100 FORMAT(' The square root of ',f5.2,' is ',f7.4)
      END
```

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 appearance of the **$LIST** metacommand

SYNTAX **$[[NO]]LIST**

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.

ACTION **$LOOPOPT** turns on compiler loop optimization; **$NOLOOPOPT** disables it

SYNTAX **$[[NO]]LOOPOPT**

REMARKS **$LOOPOPT** is the default. The **$LOOPOPT** metacommand reenables loop optimization after it has been disabled, either by the **$NOLOOPOPT** metacommand or by the /Od command-line option.

Loop optimization consists of placing invariant expressions outside the loop, and simplifying computations within the loop to speed up loop calculations.

Loop optimization is not always desirable. In the following example, the calculation of `upper` divided by `lower` is invariant, so loop optimization repositions it ahead of the loop. This means the **IF** statement can no longer check the value of the divisor before the division; if it is zero, a run-time error will occur.

```
      upper = result
      lower = quantity

      DO 200 n = 1, 100

      IF (lower .EQ. 0)
      GOTO 300 C

C     Loop optimization moves the following statement outside
C     the loop:

      factor = upper / lower
 200  CONTINUE
 300  CONTINUE
```

Due to unanticipated interactions (such as the one just described), it is dangerous to turn loop optimization on and off locally. Loop optimization should be applied to, or removed from, entire programs or procedures.

ACTION

$MESSAGE sends a character string to the standard output device during the first compiler pass

SYNTAX

$MESSAGE:*string*

Parameter	Description
string	A character constant

REMARKS

The string must be enclosed in quotes or apostrophes.

EXAMPLE

```
$MESSAGE:'Compiling program'
```

ACTION Controls the starting addresses of variables in structures

SYNTAX **$PACK[[:{1 | 2 | 4}]]**

REMARKS If there is no **$PACK** metacommand in the file, structures are packed according to the following default rules: **INTEGER*1**, **LOGICAL*1**, and all **CHARACTER** variables begin at the next available byte, whether odd or even; and all other variables begin at the next even byte. This arrangement wastes some memory space, but gives the quickest access to structure elements.

If **$PACK:1** is specified, all variables begin at the next available byte, whether odd or even. Although this slightly increases access time, no memory space is wasted.

If **$PACK:2** is specified, packing follows the default rules described above.

If **$PACK:4** is specified, **INTEGER*1**, **LOGICAL*1**, and all **CHARACTER** variables begin at the next available byte, whether odd or even; all other variables begin on four-byte boundaries.

If **$PACK** is specified (no colon or number), packing reverts to whatever mode was specified in the command-line /Zp option.

The **$PACK** metacommand may appear anywhere in a program to change structure packing as desired.

EXAMPLE

```
$PACK
.
.
.
$PACK:1
```

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 number 0C), it is the same as having a **$PAGE** metacommand before that line.

EXAMPLE

```
C     This is page one. The following metacommand starts
C     a new page in the source listing file:
$PAGE
C     This is page two. The following line starts with the ASCII
C     form-feed character, so it will also start a new page in the
C     source listing file:

C     This is page 3.
      STOP ' '
      END
```

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.

EXAMPLE

```
$PAGESIZE:60
```

ACTION **$STORAGE** allocates *n* bytes of memory for all variables declared as integer or logical variables

SYNTAX **$STORAGE:***n*

REMARKS The argument *n* must be either 2 or 4. The default is 4.

NOTE *On many microprocessors, the code required to perform 16-bit arithmetic is faster and more compact than the code required to perform 32-bit arithmetic. Therefore, unless you set the Microsoft FORTRAN* ***$STORAGE*** *metacommand to a value of 2, programs will default to 32-bit arithmetic and may run slower than expected. Setting the* ***$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 appears.

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 **$STORAGE** metacommand affects arithmetic expressions, see Section 1.7.1.2, "Type Conversion of Arithmetic Operands." For information on how the **$STORAGE** metacommand affects the passing of integer arguments, see Section 2.5, "Arguments."

EXAMPLE

```
$STORAGE:2

C     b and c are declared without a byte length, so they default
C     to the $STORAGE value of 2 bytes. a and d will be 4 bytes.

      INTEGER*4 a, d
      INTEGER   b, c

      a = 65537
      b = 1024
```

```
C       Since c is 2 bytes, it is assigned only the lower 2 bytes
C       of a+b.
        c = a + b
        d = a + b

C       The following statement produces:  1025        66561
        WRITE (*, *) c, d
        END
```

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 **$NOTSTRICT**. **$NOTSTRICT** and **$STRICT** can appear anywhere in a source file, enabling and disabling the extensions as desired.

All Microsoft FORTRAN features that are not in the FORTRAN 77 full-language standard are printed in blue. The **$STRICT** metacommand disables them.

EXAMPLE

```
$STRICT
C     The following statement produces an error, because
C     INTEGER*2 is not part of the FORTRAN 77 standard:
      INTEGER*2  i

C     The variable name (balance) will be truncated to six
C     characters:
      REAL  balance(500)

C     The following statement produces an error, because the
C     MODE= option is not part of the FORTRAN 77 standard:
      OPEN (2, FILE = 'BALANCE.DAT', MODE = 'READ')

      DO 100 i = 1, 500
C     The following statement produces an error, because the EOF
C     intrinsic function is not part of the FORTRAN 77 standard;
C     (EOF is treated as a REAL function):

         IF (EOF (2))  GOTO 200
         READ (2, '(F7.2)') balance(i)
 100  CONTINUE
 200  CONTINUE
      END
```

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 **$SUBTITLE** metacommand, the subtitle is a null string. The value of the *subtitle* string is printed in the upper-left corner of the source-listing file pages, below the title, if any appears. For a subtitle to appear on a specific page of the source-listing file, the **$SUBTITLE** metacommand must be the first statement on that page.

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
```

ACTION **$TITLE** assigns the specified title for subsequent pages of the source listing (until overridden by another **$TITLE** metacommand)

SYNTAX **$TITLE:***title*

Parameter	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
```

ACTION **$TRUNCATE** truncates all variable and program/subprogram names to six characters; **$NOTRUNCATE** disables the default or a previous **$TRUNCATE** metacommand

SYNTAX **$[NO]TRUNCATE**

REMARKS The default is **$NOTRUNCATE**. When the default is in effect, the first 31 characters in a name are significant. (Your operating system may also restrict the length of names.)

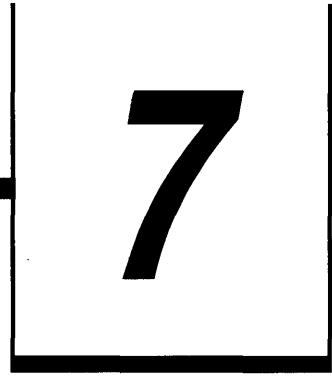
When **$TRUNCATE** or **$STRICT** is in effect, names longer than six characters generate warning messages. This can make it easier to port your code to other systems.

EXAMPLE

```
C     This program produces the following output:
C
C     74 Las Vegas Street
C     74 Las Vegas Street
C
C     Barry Floyd
C     3 Prospect Drive


      IMPLICIT CHARACTER*20 (s)

$TRUNCATE
      studentname    = 'Enrique Pieras'
      studentaddress = '74 Las Vegas Street'

      WRITE (*, 100) studentname, studentaddress

$NOTRUNCATE
      studentname    = 'Barry Floyd'
      studentaddress = '3 Prospect Drive'

      WRITE (*, 100) studentname, studentaddress

  100 FORMAT (/ 1X, A20, / 1X, A20)

      END
```

CHAPTER 7

The FL Command



The FL command automatically compiles and links your FORTRAN program in a single step. FL executes the three compiler passes, then calls LINK (the Microsoft Segmented-Executable Linker) to link the object files. Except for a few unusual situations, FL is the only command you need to compile and link your FORTRAN source files. This chapter explains how it is used.

When used by itself, the FL command creates an .EXE file that runs on most PC-compatible computers. In addition, the FL command offers a wide range of options, with which you can do the following:

- Set the memory model
- Select how floating-point calculations are performed
- Control which FORTRAN language features are available
- Create overlay files
- Create programs that run under OS/2
- Optimize the program for minimum size or maximum speed
- Assemble and link machine-language routines automatically
- Establish the search path for files
- Switch compiler warning messages on and off
- Select the page format and headers for output listings

This chapter explains what each option does and how it is used.

NOTE *This chapter assumes you have a basic knowledge of FORTRAN and know how to create and edit program files. For questions relating to the definition of the FORTRAN language, see specific chapters in this manual. The Microsoft CodeView and Utilities User's Guide explains how to use the symbolic debugger provided with this package.*

For a quick introduction to running the compiler and linker with FL, see the Microsoft FORTRAN Getting Started manual.

Section 7.1 explains how to use the FL command to produce an executable program from a FORTRAN source file. It also describes how to run the program and pass command-line arguments to it, if desired.

The remainder of the chapter discusses commonly used FL options. The FL options that control floating-point operations and memory models are discussed in *Microsoft FORTRAN Advanced Topics*. A summary of the FL commands and all available options appears in the *Microsoft FORTRAN Quick Reference Guide* provided with this package.

7.1 The Basics: Compiling, Linking, and Running FORTRAN Files

This section explains how to use FL to compile and link FORTRAN files, and it discusses the rules and conventions that apply to the file names and options used with FL. It also explains how to run the executable program created by FL.

7.1.1 Compiling and Linking with FL

The FL command takes the following form:

FL [[*option...*]] [[*filespec...*]] [[*option...*]] [[*filespec...*]]
[[/link [[*libfield*]] [[*linkoptions*]]]] [[/MA *option*]]

NOTE *Syntax examples too long to fit on one line are continued on two or more lines.*

Each *option* is one of the command-line options described in this manual, and each *filespec* names a file to be processed. The FL command automatically specifies the appropriate library to be used during linking; however, you can use the /link option with the optional *libfield* and *linkoptions* to specify additional libraries and options to be used during linking. See Section 7.32, "Using FL to Link without Compiling," for more information.

You can give any number of options and file names on the command line, provided the command line does not exceed 128 characters.

The FL command can process source files, object files, or a combination of source and object files. It uses the file-name extension (the period plus any letters following it) to determine what kind of processing the file needs, as shown below:

- If the file has a .FOR extension, FL compiles the file.
- If the file has an .OBJ extension, FL invokes the linker.
- If the extension is omitted or is anything other than .FOR or .OBJ, FL assumes the file is an object file unless the file name is part of a /Tf option. If the file name appears in a /Tf option, FL assumes the file is a FORTRAN source file. See Section 7.10 for a description of the /Tf option.

FL also accepts files with the .LIB and .DEF extensions and passes them to the linker in the correct order. Files that end with the .ASM extension or that appear in a /Ta option are passed to MASM.

You can use the DOS wild-card characters (* and ?) to process all files meeting the wild-card specification, as long as the files have the required extensions. See the DOS manual for more information on wild-card characters.

Any *filespec* on the FL command line can include a full or partial path specification, allowing you to process files in different directories or on different drives. A full path specification starts with the drive name. A partial path specification gives one or more directory names before the name of the file, but does not give a drive name. If a full path is not given, FL assumes the path starts from the current working directory.

File names can contain any combination of uppercase or lowercase letters. For example, the following three file names are equivalent:

```
abcde.for
ABCDE.FOR
aBcDe.foR
```

When FL compiles source files, it creates object files. By default, these object files have the same base names as the corresponding source files, but with the extension .OBJ instead of .FOR. (The base name of a file is the portion of the name preceding the period, but excluding the path and drive name, if any.) You can use the /Fo option to assign a different name to an object file.

These object files, along with any .OBJ files given on the command line, are linked to form an executable program file. The executable file has the base name of the first file (source or object) on the command line, plus an .EXE extension. If only .OBJ files are given on the command line, the compilation stage is skipped altogether, and the files are simply linked.

You can tell whether FL is compiling or linking by the messages on the screen. When FL invokes the compiler, a message similar to the following is written to *stderr*:

```
Microsoft (R) FORTRAN Optimizing Compiler Version 5.00
Copyright (C) Microsoft Corp 1989.  All rights reserved.
```

As each source file on the command line is compiled, its name appears on the screen. When all source files have been compiled and the linker is invoked, a message similar to the following appears:

```
Microsoft (R) Segmented-Executable Linker  Version 5.03
Copyright (C) Microsoft Corp 1984-1989. All rights reserved.
```

This message is followed by several lines showing Microsoft LINK prompts and the responses provided by FL. The FL command uses the response-file method of invoking Microsoft LINK.

NOTE *FL ignores case in the linking stage. For example, the linker regards Global and GLOBAL as the same symbol. If you do not wish to ignore case, you must include the option, /link /NOI (NOIGNORECASE). See Chapter 20, "Linking Object Files with LINK," in the Environment and Tools for more information.*

7.1.2 Using FL Options

The FL command offers many command-line options to control and modify the compiler's operation. Options begin with a slash (/) and contain one or more letters. You can use a dash (-) instead of a slash if you prefer. For example, /I and -I are both acceptable forms of the I option.

NOTE *Although file names can be either uppercase or lowercase, options must be given exactly as shown. For example, /Zd is a valid option, but /ZD and /zd are not.*

Options can appear anywhere on the FL command line. In general, an option applies to all files following it on the command line, and does not affect files preceding it. However, not all options obey this rule (see the discussion of a particular option for information on its behavior). Most FL options apply only to the compilation process. Unless otherwise noted, options do not affect object files on the command line.

Some options take arguments, such as file names, strings, or numbers. In some cases, spaces are permitted between the option and its argument; check the syntax of each option to confirm this.

Some options consist of more than one letter. When this is the case, no spaces are allowed between the letters of an option. For example, the /Sp option cannot appear as /S p or a command-line error occurs.

Chapters 1 and 2 of *Microsoft FORTRAN Advanced Topics* describe FL options for floating-point operations and memory models, respectively. Additional linking options are explained in Chapter 20 of *Environment and Tools*.

7.1.3 The FL Environment Variable

The FL environment variable is a convenient way to specify frequently used options and files without having to enter them manually. If the FL environment variable exists, the options specified in the variable are automatically added to the command line.

Use the SET command to set the FL environment variable. It is usually easier to add the SET command to the AUTOEXEC.BAT file than to enter it manually when you reboot. For example, the following SET command sets FL so programs are compiled and linked for use with the Microsoft CodeView debugger:

```
SET FL=/Zi /Od
```

All options in the FL variable are processed before the options on the command line, with the exception of /link options. After all other options are processed, any /link options in the FL variable are processed, followed by the /link options on the command line. For example, if the FL environment variable is set to `/Zi /Od /link /I`, the following two commands are equivalent:

```
FL          MAINPGM.FOR MODULE1.FOR /link     /NOP
FL /Zi /Od MAINPGM.FOR MODULE1.FOR /link /I /NOP
```

When conflicting options appear on the same command line, the last option usually takes precedence. Therefore, since options specified in the FL variable are processed first, they will be overridden by conflicting options on the command line.

7.1.4 Specifying the Next Compiler Pass

If FL cannot locate the next compiler pass it needs (there are three), it prompts you for the complete path name. (This occurs when compiling from floppy disks, or if you have changed the file name of one of the passes.) If the pass is on another disk, insert it in any available drive. Type the full path name, then press ENTER.

7.1.5 Stopping FL

If you need to abort the compiling and linking session, press CTRL+C or CTRL+BREAK. You will be returned to the DOS command level, where you can restart FL.

7.1.6 Using the FL Command (Specific Examples)

The command line below compiles the files A.FOR and B.FOR, creating object files named A.OBJ and B.OBJ:

```
FL  A.FOR  B.FOR  C.OBJ  D
```

These object files are then linked with C.OBJ and D.OBJ to form an executable file named A.EXE (since the base name of the first file on the command line is A.) Note that the extension .OBJ is assumed for D since no extension is given on the command line.

The command line below compiles all source files with the default extension (.FOR) in the current working directory:

```
FL  *.FOR
```

The resulting object files are linked to form an executable file whose base name is the same as the base name of the first file compiled.

The command below links all object files with the default extension (.OBJ) in the current working directory, creating an executable file whose base name is the same as the base name of the first object file found.

```
FL  *.OBJ
```

This will usually be the first file listed in the disk directory, and not necessarily the first file alphabetically.

7.1.7 Running Your FORTRAN Program

Compiling and linking a program produces an executable file with the extension .EXE. This file can be run from the operating system. The PATH environment variable is used to find executable files. You can execute your program from any directory, as long as the executable program file is either in your current working directory or in one of the directories given in the PATH environment variable.

The command line that executes the program can also include additional file names. These names are used to satisfy **OPEN** statements in your program that leave the file-name field blank. The first file name on the command line is used for the first such **OPEN** statement executed, the second file name is used for the second **OPEN** statement, and so on.

These command-line file names and any other arguments are also available by calling the **GETARG** procedure. You can use the command-line file names in **OPEN** statements without filenames, and still retrieve them with **GETARG**. See Chapter 5, "Intrinsic Functions and Additional Procedures," for more information. There is no interaction between these uses, and none of the names are changed or deleted.

NOTE *If your program executes a* ***READ*** *or* ***WRITE*** *statement specifying a file that has not been opened, the effect is the same as that of an* ***OPEN*** *statement with a blank file name. Default values are assigned to the parameters normally given in the* ***OPEN*** *statement. This operation is called an "implicit* ***OPEN****."*

If the file names on the command line outnumber the **OPEN** statements with blank file names, the extra file names are ignored. However, the remaining arguments can still be accessed with the **GETARG** procedure.

If more **OPEN** statements with blank file names are executed than there are file names on the command line, you will be prompted to enter a file name for each additional **OPEN** statement. You are also prompted if you give a null file name (see the example below).

Each file name on the command line must be separated from the names around it by one or more spaces or tab characters. Each name can be enclosed in quotation marks ("*filename*") if desired, but this is not required. A null argument consists of an empty set of double quotation marks, with no file name enclosed ("").

The following example runs the program MYPROG.EXE:

```
MYPROG  ""  OUTPUT.DAT
```

Since the first file-name argument is null, the first **OPEN** statement with a blank file-name field produces the following message:

```
File name missing or blank - please enter file name
UNIT number ?
```

The *number* is the unit number specified in the **OPEN** statement. The file name OUTPUT.DAT is used for the second such **OPEN** statement executed. If additional **OPEN** statements with blank file-name fields are executed, you will be prompted for more file names.

7.2 Getting Help with FL Options (/HELP)

Option

/HELP
/help

The /HELP option displays a list of the most commonly used FORTRAN options. (See the *Microsoft FORTRAN Quick Reference Guide* for a complete alphabetical listing of FL options.) For this option to work, the file containing the FORTRAN options, FL.HLP, must be in the current directory or in the path given in the PATH environment variable. If FL cannot find this file, it displays the following error message:

```
cannot open help file, 'fl.hlp'
```

This option is not case sensitive. Any combination of uppercase and lowercase letters will work.

When the /HELP option appears on the FL command line, FL displays the list of options but does not take any other action, regardless of what other options appear on the command line. For example, if you give a source-file name along with the /HELP option, FL does not compile the source file.

The help screen prompts you to press any key before returning to the operating system prompt. This keeps the top lines of the help screen in view. Once you press a key and return to the operating system prompt, the top lines scroll out of view.

Examples

The following examples show how you can save the help screen for future reference by sending it to a file or printer:

```
FL  /HELP > HELP.DOC

FL  /HELP >PRN
```

The first example above saves the help screen in a file named HELP.DOC. In the second example, the screen output is sent directly to the printer device, PRN. (See Section 7.15, "Special File Names," or your operating system documentation for a list of device names that can be used in redirection.)

Note that you may have to press ENTER several times to make sure that all help messages are saved or printed. Since the messages may be displayed on several separate screens, FL waits for you to enter a keystroke before displaying the next screenful of messages. Also, you must press an additional key (any key can be used, including ENTER) after giving the FL command, since the help screen requires you to press a key before returning to the operating system prompt.

7.3 Floating-Point Options (/FP)

The following options control the way your program handles floating-point operations:

Option	Effect
/FPi	Generates in-line instructions and selects the emulator math package
/FPi87	Generates in-line instructions and selects 8087/287/387 commands
/FPc	Generates floating-point calls and selects the emulator math package

/FPc87	Generates floating-point calls and selects 8087/80287 commands
/FPa	Generates floating-point calls and selects the alternate math package

See Chapter 1, "Controlling Floating-Point Operations," in *Microsoft FORTRAN Advanced Topics* for a complete description of these options.

By default, programs use the 8087/80287 coprocessor. If your computer does not have a coprocessor, you must specify a floating-point option that uses either the emulator or the alternate-math package. (The emulator package exactly duplicates the operations of the 8087/80287 coprocessor in software. The alternate-math package is not as accurate as the 8087/80287 emulation, nor does it adhere to the IEEE floating-point standards, but it does execute faster.)

The Microsoft FORTRAN Compiler does not generate true in-line 8087/80287 code when you use the /FPi87 option. Instead, the compiler inserts software interrupts to library code, which fixes each interrupt to use either the emulator or coprocessor, as appropriate. If you have force-linked the emulator code to your program, these interrupts will call the emulator if a math chip is not available.

If your program is designed to run only with a coprocessor, you can speed up the program by assembling and linking the code below, which eliminates the fix-up processing. In addition, all divide-by-zero errors will be masked.

```
PUBLIC  FIARQQ, FICRQQ, FIDRQQ, FIERQQ, FISRQQ, FIWRQQ
PUBLIC  FJARQQ, FJCRQQ, FJSRQQ

FIARQQ    EQU  0
FICRQQ    EQU  0
FIDRQQ    EQU  0
FIERQQ    EQU  0
FISRQQ    EQU  0
FIWRQQ    EQU  0
FJARQQ    EQU  0
FJCRQQ    EQU  0
FJSRQQ    EQU  0

extrn   __fpmath:far
extrn   __fpsignal:far
extrn   __fptaskdata:far

CDATA   segment  word  common 'DATA'
        dw   0
        dd   __fpmath
        dd   __fpsignal
        dd   __fptaskdata
CDATA   ends

end
```

7.4 Memory-Model Options (/A, /M)

The /A option specifies the program's memory model. The memory model defines the rules the compiler uses to organize the program's code and data segments in memory. The following standard memory models are available:

Option	Effect
/AL	Chooses the large memory model (default)
/AM	Chooses the medium memory model
/AH	Chooses the huge memory model

See Chapter 2, "Selecting a Memory Model," in *Microsoft FORTRAN Advanced Topics* for a more complete description of these options and the memory models they specify.

The floating-point and memory-model options you choose determine the standard library name that FL places in the object file it creates. This library is the one the linker searches for by default. Table 7.1 shows each combination of memory-model and floating-point options and the corresponding library name that FL places in the object file.

Table 7.1 FL Options and Default Libraries

Floating-Point Option	Memory-Model Option	Default Library
/FPi87 or /FPc87	/AL or /AH	LLIBFOR7.LIB
	/AM	MLIBFOR7.LIB
/FPi or /FPc	/AL or /AH	LLIBFORE.LIB
	/AM	MLIBFORE.LIB
/FPa	/AL or /AH	LLIBFORA.LIB
	/AM	MLIBFORA.LIB

The /M option supports the use of dynamic-link and multithread applications in OS/2. These switches cannot be used with the /Am option since threads and dynamic-link libraries are not supported in medium model.

Option	Use
/MT	Multithread applications
/MD	Dynamic-link library and multithread applications

When the /MT option is used, the program is automatically linked with LLIBFMT.LIB, regardless of the memory model. When the /MD option is used, all default libraries are overridden and you must create your own library, which is then specified on the command line (or when using LINK).

Both the /MT and /MD option imply the /G2 (286 instruction set) option, since multithread and dynamic-link library programs run only under OS/2. They also imply the /FPi87 (math coprocessor) option. (This default may be overridden by any other floating-point option except /FPa, which, when used, causes a command-line error.) In addition, the compiler assumes that SS is not equal to DS (the stack may be in a segment other than the default data segment).

Both the /MT and /MD options are incompatible with /Fb (bound) and /Lr or /Lc (real mode), since threaded and dynamic-link library applications are assumed to be unbound programs running in protected mode.

NOTE *If you renamed any of the libraries created while running SETUP, the library name embedded in the object file might not match the renamed library (as, for example, when you use the new default names). In these cases, you must explicitly specify the new library name to the linker. See Section 7.30, "Linking with Libraries," for more information.*

7.5 Library Options (/Lp, /Lr, /Lc)

Microsoft FORTRAN supports the creation of applications that run in both DOS (real-mode) and OS/2 (protected-mode), each with its own set of run-time libraries.

When you install FORTRAN, the SETUP program gives you the option to rename libraries to the default names used by the compiler. If during SETUP you chose not to rename a set of libraries, the following options can be used to specify libraries when compiling a program:

Option	Effect
/Lp	Specifies protected-mode libraries
/Lr	Specifies real-mode libraries
/Lc	Same as /Lr, but used by IBM C compiler; provided for compatibility

You need to specify a /L option when you install FORTRAN for either DOS or OS/2 mode (but not both) and chose not to rename your libraries, or you install FORTRAN for both DOS and OS/2.

SETUP normally gives each combined library a name of the form *m*LIBF*fs*.LIB, where *m* is M or L (Medium or Large memory model); *f* is A, E, or 7 (alternate, emulator, or 80x87 coprocessor math library); and *s* is R (DOS real mode) or P (OS/2 protected mode).

The FORTRAN compiler inserts the name of a default library in every object file so the linker knows which library to use. This default library name is of the form *m*LIBFOR*f*.LIB, and does not have the R or P specifier.

If you install FORTRAN for only DOS or only OS/2, you should use the default library names. If you install for both modes, both sets of libraries cannot have identical names, so only one set can use the default names. The other will contain R or P, and you must specify the /L option to use the proper naming convention for this set.

As an example, if you installed for both DOS and OS/2, chose the Large model, coprocessor library, and chose to rename your DOS library, SETUP would build two libraries: LLIBFOR7.LIB for DOS, and LLIBF7P.LIB for OS/2. To compile and link the program QSORT.FOR for OS/2, you would type:

```
FL /Lp QSORT.FOR
```

using /Lp to specify the LLIBF7P.LIB library. No /L switch is required for DOS since LLIBFOR7.LIB has the default name.

NOTE *The /L option does not change the name of the default library the compiler places in an object file. Rather, it tells the linker to ignore this default name and supplies the name of the appropriate replacement library. If you do not use the FL utility to link your program, the /L option will have no effect. In this situation, you must explicitly name a library on the LINK command line. See Section 7.30 for more information on linking with libraries.*

7.6 Data Threshold Option (/Gt)

Option

/Gt[[*number*]]

The /Gt option sets the data threshold, a cutoff value the compiler uses when allocating data.

In the medium memory model, the compiler allocates all static and global data items within the default data segment. In the large and huge models, the

compiler allocates only initialized static and global data items to the default data segment.

The /Gt option can be used only with large- and huge-model programs, since medium-model programs have only one data segment. This option is particularly useful with programs that have more than 64K of initialized static and global data as small data items.

The /Gt option causes all data items whose size is greater than or equal to *number* bytes to be allocated to a new data segment outside the default data segment that is accessed with a far address. When the /Gt option is omitted, the default threshold value is 32,767 bytes. When the /Gt option is given, but no *number* is specified, the default threshold value is 256 bytes. When *number* is specified, it must immediately follow /Gt, with no intervening spaces.

7.7 Naming and Organizing Segments (/ND, /NM, /NT)

Options

/NT *textsegment*
/ND *datasegment*
/NM *textsegment*

An object-code file is sometimes referred to as a "module." A module can contain a main program, one or more procedures, or any combination of a main program and procedures.

A "segment" is a contiguous block of code or data produced by the compiler. Every module has at least two segments: a text segment containing the program code (machine instructions), and a data segment containing the program data.

All segments have names. Text segment and data segment names are normally created by the compiler. These default names depend on the memory model chosen. For example, in medium-model programs, the text segment from each is placed in a separate segment with a distinct name, formed from the module base name and the suffix _TEXT. The single data segment is named _DATA.

In large- and huge-model programs, the text and data from each module are loaded into separate segments with distinct names. Each text segment is given the name of the module plus the suffix _TEXT. The data from each segment is placed in a private segment with a unique name (except for initialized global and static data that are placed in the default data segment). The naming conventions for text and data segments are summarized in Table 7.2.

Table 7.2 Segment-Naming Conventions

Model	Text	Data	Module
Medium	*module*_TEXT	_DATA	*filename*
Large	*module*_TEXT	_DATA[1]	*filename*
Huge	*module*_TEXT	_DATA[1]	*filename*

[1] Name of default data segment; other data segments have unique private names.

The linker uses segment names to define the order in which the segments of the program appear in memory when loaded for execution. (The segments in the group named DGROUP are an exception.) Segments with the same name (both module and suffix) are loaded into the same physical memory segment.

The /NT (nametext) and /ND (namedata) options override the default names supplied by the compiler. This allows you, for example, to give two different modules the same text-segment name, ensuring that both are loaded into the same memory segment. This is useful when you wish to give a procedure the **NEAR** attribute, because it means the contents of the code segment (CS) register need not be changed when the procedure is called.

If you change the name of the default data segment using /ND, there is no guarantee the data segment (DS) register will have the same value as the stack segment (SS) register. In this case, the compiler automatically generates code to load the DS register with the correct data-segment value on entry to the code and restores the previous value upon exit.

The *textsegment* and *datasegment* arguments can be any combination of letters and digits.

In previous versions of FL, the /NM option changed the name of a module. In the version supplied with FORTRAN 5.0, /NM is identical to the /NT option.

7.8 Creating Bound Program Files (/Fb)

Option	Effect
/Fb	Creates a bound .EXE file with the source file base name
/Fb*filename*	Creates a bound file named *filename*.EXE and a protected-mode .EXE file with the source file base name

To create a bound program that can run under OS/2 or DOS, use a command of the following type:

```
FL  /Lp  /Fb  QSORT.FOR
```

To create both a protected-mode .EXE file and a bound .EXE file, specify a file name that is different from the base name of the source file:

```
FL  /Lp  /FbQSORTB  QSORT.FOR
```

In the preceding example, the protected-mode program QSORT.EXE is created first, then the bound program QSORTB.EXE is created. If the name given in the /Fb option is the same as the base name of the source file, the bound program file overwrites the protected-mode program that was created first.

In order for the /Fb option to work, the files DOSCALLS.LIB, APILMR.OBJ, and API.LIB must be in the current directory or in one of the directories specified in the LIB environment variable.

The file APILMR.OBJ is automatically bound with the program. If your medium-model program does not use allocatable arrays, and if you do not wish to reserve a full 64K DGROUP, run the BIND command manually, specifying the /n switch and omitting APILMR.OBJ. See the *Microsoft CodeView and Utilities User's Guide* for information about the BIND command.

If you use the /Fm (map) option with the /Fb option, the resulting map file contains the map from BIND, rather than the linker map.

When you bind mixed-language executable files that have non-Family API calls, you must execute the BIND command directly and specify the /n switch. See Chapter 3, "Mixed-Language Programming," of *Microsoft FORTRAN Advanced Topics* for more information.

7.9 FORTRAN-Specific Options (/4Y, /4N)

The following options begin with the numeral 4, indicating they are specific to FORTRAN 77. The letter Y (yes) indicates the option is enabled, and the letter N (no) indicates it is disabled.

More than one /4 option may appear on the command line. An option applies to all the files succeeding it, until a conflicting or canceling option removes its effect for any remaining files. A contradictory metacommand within a file overrides the effect of a command-line option for that file only.

The /4 options may be combined. For example, /4Ysfb disables all Microsoft FORTRAN extensions, permits free-form entry, and enables extended error handling. Note that groups of Y and N options must appear as separate command-line options; they cannot be mixed in a single option.

7.9.1 Controlling Optional Language Features (/4Ys, /4Yi, /4Yv)

Option	Effect
/4{Y | N}s	Disables (Y) or enables (N) all Microsoft FORTRAN extensions
/4{Y | N}i	Enables (Y) only SAA extensions; disables (N) full Microsoft FORTRAN extensions
/4{Y | N}v	Enables (Y) only VAX extensions; disables (N) full Microsoft FORTRAN extensions

Microsoft FORTRAN includes many optional features that are not part of ANSI-standard FORTRAN. All these extensions are normally available by default. To disable all of them, use the /4Ys command-line switch. Any language feature which is not part of the strict FORTRAN 77 definition is then tagged by the compiler as an error. (The /4Ys option is equivalent to the **$STRICT** metacommand at the beginning of the file.)

The /4Ys option applies to all source-code files following it on the command line, unless the /4Ns option disables it for any of the remaining files. The **$NOTSTRICT** metacommand within a file overrides the /4Ys command option for that file only. The **$STRICT** metacommand within a file overrides the /4Ns command option for that file only.

Many Microsoft FORTRAN extensions fall into one of two categories: IBM SAA (Systems Application Architecture) extensions, and DEC® VAX extensions.

To simplify porting code from an SAA or VAX environment to a personal computer, you can disable all Microsoft FORTRAN extensions except the SAA extensions, or disable all Microsoft FORTRAN extensions except the VAX extensions. (The compiler recognizes only the strict FORTRAN 77 language plus either the SAA or the VAX extensions, but not both.)

This is done with the /4Yi and /4Yv options, respectively. Either option applies to all files following it on the command line, until the /4Ni or /4Nv option reverts to using only Microsoft FORTRAN extensions.

Microsoft FORTRAN includes all IBM SAA extensions, which are listed below:

- 31-character names
- Bit-manipulation intrinsic
- Case-insensitive source
- **COMMON** allows character and noncharacter in same block
- **CONJG**, **HFIX**, and **IMAG** intrinsic functions

- Data initialization in type statements
- **EQUIVALENCE** allows association of character and noncharacter
- Functions **IOR**, **IAND**, **NOT**, **IEOR**, **ISHFT**, **BTEST**, **IBSET**, and **IBCLR**
- **IMPLICIT NONE**
- **INCLUDE** compiler directive
- **INTEGER*2**, **COMPLEX*16**, and **LOGICAL*1** data types
- Optional length specifications in **INTEGER**, **REAL**, **COMPLEX**, and **LOGICAL** type statements
- Use of underscore (_) in names
- Z edit descriptor

Microsoft FORTRAN includes many (but not all) VAX extensions. The supported extensions are:

- 31-character names
- **ACCESS** selector **'APPEND'** in the **OPEN** statement
- Allowing integer arrays to contain **FORMAT** statements
- **BLOCKSIZE** and **NML** I/O keywords
- Debug comment lines
- **DO** statements without specified labels
- **DO WHILE** statement
- **END DO** statement
- **EQUIVALENCE** of character and noncharacter items
- **EQUIVALENCE** to a multi-dimensioned array with only one subscript
- Exponentiation of **REAL** and **COMPLEX** statements
- **IMPLICIT NONE**
- **INCLUDE** compiler directive
- Initialization on the declaration line
- In-line comments
- Length specification within the **FUNCTION** statement
- Length specifications within type declarations

- Mixing of character and noncharacter items in **COMMON** statements
- Noninteger alternate return values
- Noninteger array subscripts
- Numeric operands for **.AND.**, **.OR.**, etc.
- Specified common-block variables in **DATA** statements outside a **BLOCK DATA** subroutine
- **STRUCTURE**, **UNION**, **MAP** statements
- Up to 99 continuation lines
- Use of dollar sign ($) in names
- **.XOR.** operator

7.9.2 Controlling Source-File Syntax (/4Yf, /4Nf, /4Yt, /4Nt, /4Y6, /4N6)

Option	Effect
/4{Y | N}f	Enables (Y) or disables (N) free-form format
/4{Y | N}t	Enables (Y) or disables (N) truncation of variable names
/4{Y | N}6	Enables (Y) or disables (N) FORTRAN 66-style **DO** statements

These options control the structure of a FORTRAN source file and two elements of FORTRAN syntax. They correspond to FORTRAN metacommands which are described in detail in Chapter 6, "Metacommands." The following list gives the metacommand corresponding to each option and identifies the option's default:

Option	Metacommand
/4Yf	**$FREEFORM**
/4Nf (default)	**$NOFREEFORM**
/4Yt	**$TRUNCATE**
/4Nt (default)	**$NOTRUNCATE**
/4Y6	**$DO66**
/4N6 (default)	None

Each option has the same effect as placing the corresponding metacommand at the beginning of the source file. A conflicting metacommand in the source file overrides the command-line option for that file only. No error occurs if an option is used with a file that already contains the corresponding metacommand.

Examples

```
FL  /c  /4Yds  TEST.FOR  /4Nd  STABLE.FOR
```

The command line above causes FL to compile TEST.FOR using truncation (disallowing all Microsoft extensions). The declare option is also enabled, so use of undeclared variables produces warning messages. When the second file, STABLE.FOR, is compiled, the truncation is still in effect, but the declare option is disabled.

```
FL  /4Yf  /4Nt  *.FOR
```

The command line above enables free-form format and disables truncation of variable names when compiling and linking all source files in the current working directory.

7.9.2.1 The Debug Option

Option

/4{Y | N}b

The debug option controls extended error handling at run time. It provides information to be used by the error-handling system in the program file. See the discussion of the **$DEBUG** metacommand in Chapter 6 for a description of the types of errors that are detected in extended error handling. When the debug option is enabled, loop optimization in the program is disabled.

Debugging is enabled by giving the /4Yb option, and disabled with /4Nb. By default, debugging is disabled.

The debug option has the same effect as a **$DEBUG** or **$NODEBUG** metacommand at the beginning of the source file being compiled. If a **$DEBUG** or **$NODEBUG** metacommand appears later in the file being compiled, debugging for that file is enabled or disabled, as appropriate.

The /4Yb option does not accept a string argument for conditional compilation. Use the /4cc option, described in Section 7.9.5, instead.

Examples

```
FL  MAIN.FOR  /4Yb  /Fs  TEST.FOR
```

The example above compiles and links two files. Debugging is enabled for TEST.FOR, and a source listing named TEST.LST is created. Neither the debugging option nor the source-listing option applies to MAIN.FOR.

```
FL  /c  /4Ybd  ONE.FOR  /4Nd  TWO.FOR
```

The example above compiles ONE.FOR with both the debug and declare options enabled. (The following section describes the declare option.) The declare option is disabled when compiling TWO.FOR, but the debug option is still in effect.

NOTE *When using the /4Yb option, it is recommended that you also use the /Ge option to enable stack probes. See Section 7.25 for more information on /Ge.*

7.9.2.2 The Declare Option

Option

/4{Y | N}d

The declare option controls warnings about undeclared variables. When the declare option is enabled, the compiler generates a warning message at the first use of any variable which has not been declared in a type statement.

The /4Yd option enables the declare option and /4Nd disables it. The declare option is disabled by default (unless a **$DECLARE** metacommand is in the source file).

The declare compiler option has the same effect as a **$DECLARE** metacommand or a **$NODECLARE** metacommand at the beginning of each source file. If **$DECLARE** or **$NODECLARE** metacommands appear later in the file, warnings are enabled or disabled, as appropriate. Note that if the source file contains a **$DECLARE** or **$NODECLARE** metacommand at the beginning, the /4Yd or /4Nd option has no effect.

Examples

```
FL  /4Ybd  *.FOR > DECLARE
```

The example above compiles and links all source files with the default extension (.FOR) in the current working directory. The debug and declare options are both enabled. All messages (including warnings about undeclared variables) are redirected to the file DECLARE.

```
FL  /4Yb  ONE.FOR  /4Yd  TWO.FOR
```

The example above turns on debugging for both ONE.FOR and TWO.FOR; the declare option is also enabled for TWO.FOR.

7.9.3 Automatic Variables

Option

/4{Y | N}a

The /4Ya option causes all eligible variables in the succeeding files to be automatic. They are declared on the stack. (In Microsoft FORTRAN, the default is that all variables are static. They have defined memory locations.) The /4Na option disables automatic variables (except as declared within the program) for succeeding files. For more information, see the **AUTOMATIC** entry in Section 4.2, "Statement Directory."

7.9.4 Setting the Default Integer Size (/4I2, /4I4)

Option

/4I{ 2 | 4}

The /4I option allocates either 2 or 4 bytes of memory for all variables declared in the source file as **INTEGER** or **LOGICAL** variables. The default allocation is 4 bytes. The /4I option applies to the remainder of the command line or until another /4I option appears.

Example

```
FL  /4I2  /FeTESTPROG  *.FOR
```

This example allocates 2 bytes of memory for **INTEGER** and **LOGICAL** variables when compiling and linking all source files in the current working directory. The executable file is named TESTPROG.EXE.

The /4I option has the same effect as a **$STORAGE** metacommand at the top of each file that is being compiled. If a **$STORAGE** metacommand already appears in the file being compiled, the size given by the metacommand is used. The **$STORAGE** metacommand in a particular file affects only that file and does not change the effects of /4I on any other files on the command line. See Chapter 6,"Metacommands," for more information on the **$STORAGE** metacommand.

7.9.5 Conditional Compilation (/4cc, /D)

Options

/4cc*string*
/D*symbol*[[=*val*]]

The /4cc option permits conditional compilation of a source file. The *string* is a set of alphabetic characters controlling which lines in the source file are to be compiled.

Any line with a letter in column 1 found in *string* is compiled; lines beginning with other letters are treated as comments. All lines not beginning with letters are compiled normally. Case is not significant. The letter must appear in column 1 of the source-file line.

The *string* can be enclosed in quotation marks ("") if desired, but the quotation marks are not required.

NOTE *Program lines with the character* `C` *or* `c` *in column 1 are always treated as comments.*

Example

```
FL  /c  /4ccXYZ  PRELIM.FOR
```

In this example, all lines in the file PRELIM.FOR beginning with `X`, `Y`, or `Z` are compiled.

The /D option defines a symbolic name used by the conditional compilation metacommands. It is equivalent to the **$DEFINE** metacommand within the program file. The name must immediately follow /D, with no intervening space.

An equal sign and integer value may be added. This assigns a specific value to the symbol that the conditional-compilation metacommands can check.

Examples

```
FL /Dtestprocs foo.for
FL /Dwhichpart=4 foo.for
```

In the first example, the symbol `testprocs` is defined. The compiler will then evaluate any **DEFINED** `(testprocs)` expression in the conditional-compilation metacommands as true (unless an `$UNDEFINE testprocs` metacommand in the program file cancels the definition).

In the second example, the symbol `whichpart` is defined and given the integer value four.

7.10 Specifying Source Files (/Tf, /Ta)

Options

/Tf [ ]*sourcefile*
/Ta [ ]*sourcefile*

The FL command assumes that any filename ending in .FOR is a FORTRAN source file, and invokes the compiler. Likewise, the FL command assumes that any filename ending in .ASM is an assembly source file, and tries to invoke the

Microsoft assembler. If your files do not have these identifying extensions, you can use the /Tf and /Ta options to specify that the following filename is a FORTRAN or assembly file, respectively.

If you have to specify more than one source file with an extension other than .FOR or .ASM, it is safest to list each source file in a separate /Tf or /Ta option. Although a *sourcefile* with a wild-card character is legal, this use of wild-card characters may cause problems. If a *sourcefile* with a wild-card character represents a single file, then FL behaves as expected: it considers that single file to be a FORTRAN (or assembly-language) source file. However, if a *sourcefile* with a wild-card character represents more than one file, FL treats only the first file as a FORTRAN (or assembly-language) source file. It treats any other files that *sourcefile* represents as object files. Masm must be in the path where the FL command can find it.

Examples

```
FL  MAIN.FOR  /TfTEST.PRG  /TfCOLLATE.PRG  PRINT.PRG
```

In the example above, the FL command compiles the three source files MAIN.FOR, TEST.PRG, and COLLATE.PRG. Since the file PRINT.PRG is given without a /Tf option, FL treats it as an object file. Thus, after compiling the three source files, FL links the object files MAIN.OBJ, TEST.OBJ, COLLATE.OBJ, and PRINT.PRG.

```
FL  /TfTEST?.F
```

For the example above, assume the files TEST1.F, TEST2.F, and TEST3.F all exist in the current directory. In this example, the FL command would compile TEST1.F as a FORTRAN program and then try to treat TEST2.F and TEST3.F as object files. The FL command shown above would have the same effect as the following FL command:

```
FL  /TfTEST1.F  TEST2.F  TEST3.F
```

If your program is compiled using the /Tf option, you must explicitly select the FORTRAN expression evaluator when using the CodeView debugger. CodeView only uses the FORTRAN evaluator as the default when the source file has the .FOR extension.

The /Ta option is used with the /MA option. The /MA option specifies that the following file is a Microsoft Assembler (MASM) file, and the file is to be assembled and linked with the program. If the file name does not end in .ASM, you must use the /Ta option to specify the full file name. The space between /Ta and *sourcefile* is optional.

7.11 Compiling without Linking (/c)

Option

/c

The /c (compile-only) option suppresses linking. Source files are compiled, but the resulting object files are not linked, no executable file is created, and any object files specified are ignored. This option is useful when you are compiling individual source files that do not make up a complete program.

The /c option applies to the entire FL command line, regardless of the option's position on the command line.

Example

```
FL  /c *.FOR
```

This command line compiles, but does not link, all files with the extension .FOR in the current working directory.

7.12 Naming the Object File (/Fo)

Option

/Fo*objfile*

By default, FL gives each object file the same base name as the corresponding source file, plus the extension .OBJ. The /Fo option gives an object file a different name or creates it in a different directory.

The *objfile* argument must appear immediately after the option, with no intervening spaces. The *objfile* argument can be a file specification, a drive name, or a path specification.

If *objfile* is a file specification, the /Fo option applies only to the source file immediately following the option on the command line. The object file created by compiling that source file has the name given by *objfile*.

If *objfile* is a drive name or path specification, the FL command creates object files in the given location for every source file following the /Fo option on the command line. The object files take their default names (each object file has the base name of the corresponding source file).

NOTE *When you give only a path specification, the objfile argument must end with a backslash (\) so that FL can distinguish between it and an ordinary file name.*

You can use any name or extension you like for *objfile*. However, it is recommended that you use the conventional .OBJ extension because the FL command, as well as the LINK and LIB utilities, expects .OBJ when processing object files. If you choose an object-file name that lacks an extension, FL automatically adds the .OBJ extension. However, if you pick a name with a blank extension (a name ending in a period), FL does not add an extension.

Examples

```
FL  /c  /FoSUB\THAT  THIS.FOR
```

The example above compiles the file THIS.FOR and creates an object file named THAT.OBJ in the subdirectory `SUB`. Note that FL automatically appends the .OBJ extension. Linking is suppressed because the /c option is given.

```
FL  /FoB:\OBJECT\  *.FOR
```

The example above compiles and links all source files with the extension .FOR in the current working directory. The option `/FoB:\OBJECT\` tells FL to create all the object files in the directory named OBJECT on drive B. Each object file has the base name of the corresponding source file, plus the extension .OBJ.

7.13 Naming the Executable File (/Fe)

Option

/Fe*exefile*

By default, the executable file produced by the FL command is given the base name of the first file (source or object) on the command line, plus the extension .EXE. The /Fe option gives the executable file a different name or creates it in a different directory.

Since only one executable file is created, it does not matter where the /Fe option appears on the command line. If the /Fe option appears more than once, the last name prevails.

/Fe applies only during linking. If /c suppresses linking, /Fe has no effect.

The *exefile* argument must appear immediately after the option, with no intervening spaces. The *exefile* argument can be a file specification, a drive name, or a path specification.

If *exefile* is a file specification, the executable file is given the specified name. If *exefile* is a drive name or path specification, the FL command creates the executable file in the given location, using the default name (the base name of the first file plus .EXE).

NOTE *When you give a path specification as the exefile argument, the path specification must end with a backslash (\ ) so that FL can distinguish it from an ordinary file name.*

You can give any name or extension you like for the *exefile*. If you give a file name without an extension, FL automatically appends the .EXE extension. If you pick a name with a blank extension (a name ending in a period), FL does not add an extension.

Examples

```
FL  /FeC:\BIN\PROCESS  *.FOR
```

The example above compiles and links all source files in the current working directory with the extension .FOR. The resulting executable file is named PROCESS.EXE and is created in the directory C:\BIN.

```
FL  /FeC:\BIN\  *.FOR
```

The example above is similar to the first example except that the executable file, instead of being named PROCESS.EXE, is given the same base name as the first file compiled. The executable file is created in the directory C:\BIN.

7.14 Creating Listing Files (/F)

The FL command offers a number of listing options. You can create a source listing, a map listing, or one of several kinds of object listings. You can also set the title and subtitle of the source listing and control the length of source-listing lines and pages.

The FL command optimizes by default, so object listings reflect the optimized code. Since optimization may involve rearranging the code, the correspondence between your source file and the machine instructions may not be clear, especially when you use the /Fc option to mingle the source and assembly code. To suppress optimization, use the /Od option, discussed in Section 7.22.

NOTE *Listings may contain names that begin with one or more underscores (for example, __chkstk) or that end with the suffix QQ. Names that use these conventions are reserved for internal use by the compiler.*

The following is a list of options that produce listings and control their appearances:

Option	Effect
/Fs[[*listfile*]]	Produces source listing
/Fl[[*listfile*]]	Produces object listing

/Fa[[*listfile*]]	Produces assembly listing
/Fc[[*listfile*]]	Produces combined source and object listing
/Fm[[*mapfile*]]	Produces map file that lists segments in order

This section describes how to use command-line options to create listings. For an example of each type of listing and a description of the information it contains, see Section 7.18, "Formats for Listings."

The options in this section require the *listfile* or *mapfile* argument (if given) to follow the option immediately, with no intervening spaces. The *listfile* can be a file specification, a drive name, or a path specification. It can also be omitted.

NOTE *When you give only a path specification as the listfile argument, the path specification must end with a backslash (\ ) so that FL can distinguish it from an ordinary file name.*

When you give a drive name or path specification as the argument to a listing option, or if you omit the argument altogether, FL uses the base name of the source file, plus the default extensions, to create a default file name. Table 7.3 lists the default file names and default extensions, which are used when you give a filename argument that lacks an extension.

Table 7.3 Default File Names and Extensions

Option	Listing Type	Default File Name[1]	Default Extension[2]
/Fs	Source	Base name of source file plus .LST	.LST
/Fl	Object	Base name of source file plus .COD	.COD
/Fa	Assembly	Base name of source file plus .ASM	.ASM
/Fc	Combined source-object	Base name of source file plus .COD	.COD
/Fm	Map	Base name of first source or object file on the command line plus .MAP	.MAP

[1] The default file name is used when the option is given with no argument, or with a drive name or path specification as the argument.

[2] The default extension is used when a file name lacking an extension is given.

Since you can process more than one file at a time with the FL command, the order in which you give listing options and the kind of argument you give for

each option (file specification, path specification, or drive name) are significant. Table 7.4 summarizes the effects of each option with each type of argument.

Table 7.4 Arguments to Listing Options

Option	File-Name Argument	Drive-Name or Path Argument[1]	No Argument
/Fa, /Fc, /Fl, /Fs	Creates a listing for next source file on command line; uses default extension if no extension is supplied	Creates listings in the given location for every source file listed after the option on the command line; uses default names	Creates listings in the current directory for every source file listed after the option on the command line; uses default names
/Fm	Uses given file name for the map file; uses default extension if no extension is supplied	Creates map file in the given directory; uses default name	Uses default name

[1] When you give just a path specification as the argument, the path specification must end with a backslash (\ ) so that FL can distinguish it from an ordinary file name.

If a source file includes one or more **$NOLIST** metacommands, the portion of that source file between each **$NOLIST** metacommand and the following **$LIST** metacommand (if any) is omitted from the listing.

Only one variation of the object or assembly listing can be produced for each source file. The /Fc option overrides the /Fa and /Fl options; whenever you use /Fc, a combined listing is produced. If you apply both the /Fa and the /Fl options to one source file, only the last listing specified is produced.

The map file is produced during the linking stage. If linking is suppressed with the /c option, the /Fm option has no effect.

7.15 Special File Names

You can use the DOS device names listed below as file-name arguments to the listing options. These special names send listing files to your terminal or printer.

Name	Device
AUX	An auxiliary device.
CON	The console (terminal).
PRN	The printer.
NUL	Specifies a null (nonexistent) file. Giving **NUL** as a file name means that no file is created.

Even if you add device designations or file-name extensions to these special file names, they remain associated with the devices listed above. For example, `A:CON.tuv` still refers to the console and is not the name of a disk file.

NOTE *When using device names, do not add a colon. The Microsoft FORTRAN Compiler does not recognize the colon. For example, use* `CON` *or* `PRN`*, not* `CON:` *or* `PRN:`.

7.16 Line Size (/Sl) and Page Size (/Sp)

Options

/Sl[[ ]]*linesize*
/Sp[[ ]]*pagesize*

The /Sl and /Sp options control the line size and page size of source listings, respectively. The default line size for source listings is 79 columns, and the page size is 63 lines. These options are useful for preparing source listings that will be printed on a printer that uses nonstandard page sizes. They are valid only if you also specify the /Fs option on the FL command line. The space between /Sl and *linesize*, or /Sp and *pagesize*, is optional.

The *linesize* argument gives the width of the listing line in columns (on line printers, columns usually correspond to characters). The number given must be a positive integer between 79 and 132, inclusive; any number outside this range produces an error message. Any line that exceeds the listing width is truncated.

The *pagesize* argument gives the number of lines on each page of the listing. The minimum number is 15; if a smaller number is given, an error message appears and the default page size is used.

The /Sl or /Sp option applies to the remainder of the command line or until the next occurrence of /Sl or /Sp on the command line. These options do not cause source listings to be created. They take effect only when the /Fs option is also given to create a source listing.

You can use metacommands in the source file to override the /Sl and /Sp options. These options have the same effects as **$LINESIZE** and **$PAGESIZE** metacommands at the top of each file being compiled. If additional **$LINESIZE** or **$PAGESIZE** metacommands appear in the file being compiled, the line size or page size for that file is changed accordingly.

The **$LINESIZE** or **$PAGESIZE** metacommands in a particular file affect only that file and do not change the effects of /Sl or /Sp on any other files on the command line.

Examples

```
FL  /c  /Fs  /Sl90  /Sp70  *.FOR
```

The example above compiles all source files with the default extension (.FOR) in the current working directory, creating a source-listing file for each source file. Each page of the source listing is 90 columns wide and 70 lines long.

```
FL  /Fs  /Sp70  MAIN.FOR  /Sp63  SUB1.FOR  SUB2.FOR
```

The example above compiles and links three source files, creating an executable file named MAIN.EXE. Three source listings are created: each page of MAIN.LST is 70 lines long, while each page of SUB1.LST and SUB2.LST is 63 lines long.

7.17 Titles (/St) and Subtitles (/Ss)

Options

/St[ ]"*title*"
/Ss[ ]"*subtitle*"

The /St and /Ss options set the title and subtitle, respectively, for source listings. The quotation marks around the *title* or *subtitle* argument can be omitted if the title or subtitle does not contain tabs or spaces. The space between /St and "*title*", or /Ss and "*subtitle*", is optional.

The *title* appears in the upper-left corner of each page of the source listing. The *subtitle* appears below the title.

The /St or /Ss option applies to the remainder of the command line or until the next occurrence of /St or /Ss on the command line. These options do not cause source listings to be created. They take effect only when the /Fs option is also used to create a source listing.

Both the /St and /Ss options can be overridden by metacommands in the source file. These options have the same effect as **$TITLE** and **$SUBTITLE** metacommands at the beginning of the file being compiled. If additional **$TITLE** or **$SUBTITLE** metacommands appear in the file, the title or subtitle is changed accordingly.

The **$TITLE** or **$SUBTITLE** metacommands in a particular file affect only that file and do not change the effects of /St or /Ss on any other files on the command line.

Examples

```
FL  /St"INCOME TAX"  /Ss15-APR  /Fs  TAX*.FOR
```

The example above compiles and links all source files beginning with TAX and ending with the default extension (.FOR) in the current working directory. Each page of the source listing contains the title `INCOME TAX` in the upper-left corner. The subtitle `15-APR` appears below the title on each page.

```
FL  /c  /Fs  /St"CALC PROG"  /Ss"COUNT"  CT.FOR  /Ss"SORT"
SRT.FOR
```

The example above compiles two source files and creates two source listings. Each source listing has a unique subtitle, but both listings have the title `CALC PROG`.

7.18 Formats for Listings

This section describes and gives examples of the five types of listings available with the FL command. See Section 7.14, "Creating Listing Files," for information on how to create these listings.

Source Listing

Source listings are useful in debugging programs during development. These listings can also document the structure of a finished program.

The source listing contains the numbered source-code lines of each procedure in the source file, along with expanded include files and any error messages. If the source file compiles with no errors more serious than warnings, the source listing also includes tables of local symbols, global symbols, and parameter symbols for each procedure. If the compiler is unable to finish compilation, it does not generate symbol tables.

At the end of the source listing is a summary of segment sizes. This summary is useful when analyzing memory requirements.

Error messages appear in the listing after the line that caused the error, as shown in the following example:

```
9    hyp = SQRT ((sidea**2) + (sideb**2)
***** sqroot.for(9) : error F2115: syntax error
```

The line number in the error message corresponds to the number of the source line immediately above the message in the source listing.

The example below shows the source listing for a simple FORTRAN program:

```
PAGE   1

10-17-86

18:20:36

  Line#  Source Line            Microsoft FORTRAN Compiler Version 5.00

       1   common a
       2   dimension a(10)
       3   real x
       4   complex c
       5   real *8 d
       6   complex *16 e
       7   character *50 f
       8   integer*2 j
       9   parameter (d=123456789.00056, e=-(.00000122, 1234354 e5))
      10   parameter (f='Note that character strings will be truncated')
      11   parameter (x=1.2345)
      12   parameter (c=(.12345, 123456.789), i = 123, j = 100)
      13   end

main  Local Symbols

Name                 Class   Type              Size   Offset

A . . . . . . . . . . COMMQQ  REAL*4              40    0000


Parameter Symbols        Type                Value

X . . . . . . . . . . REAL*4             1.2345001E+000
C . . . . . . . . . . COMPLEX*8          ( 1.2345000E-001,
                                           1.2345679E+005)
D . . . . . . . . . . REAL*8             1.2345679E+008
E . . . . . . . . . . COMPLEX*16         (-1.2199999E-006,-1.2343540E+011)
F . . . . . . . . . . CHARACTER          Note that character
                                         strings will
J . . . . . . . . . . INTEGER*2                      100
I . . . . . . . . . . INTEGER*4                      123


Global Symbols

Name                      Class   Type              Size   Offset

COMMQQ. . . . . . . . . . common                     40    0000
main. . . . . . . . . . . FSUBRT  ***               ***    0000

Code size = 0018 (24)
Data size = 0000 (0)
Bss size  = 0000 (0)

No errors detected
```

The Name column lists each global symbol, external symbol, and statically allocated variable declared in the source file. The Parameter Symbols column lists each symbolic constant defined in a **PARAMETER** statement.

For items other than functions and subroutines, the Class column contains either global, local, equiv, common, or extern, depending on how the symbol was defined. For functions and subroutines, the Class column contains the abbreviations in the following list:

Type	Abbreviation
Far function	**FFUNCT**
Near function	**NFUNCT**
Far subroutine	**FSUBRT**
Near subroutine	**NSUBRT**

The Type column lists a simplified version of the symbol's type as declared in the source file. The Type entry for functions is the type declared in the source file.

The Size column is used only for variables. This column specifies the number of bytes allocated for the variable. Note that the allocation for an external array may be unknown, so its Size field may be undefined.

The Offset column is used only for symbols with a global or local entry in the Class field. For variables, the Offset column gives the variable's relative offset in the logical data segment for the program file. (Even if a program file contains more than one compilation unit, all data is allocated in the same data segment.) Since the linker generally combines several logical data segments into a physical segment, this number is useful only for determining a variable's relative storage position.

The Value field appears only for parameter symbols. It lists the value of each symbolic constant. Character constants longer than 33 characters are truncated.

The last table in the source listing shows the segments used and their size, as shown below:

```
Code size = 0095 (149)
Data size = 003c (60)
Bss size  = 0000 (0)
```

The byte size of each segment is given first in hexadecimal, and then in decimal (in parentheses). For information on specific segment usage by different memory models, see Chapter 2, "Selecting a Memory Model," in *Microsoft FORTRAN Advanced Topics*.

Object-Listing File

The object-listing file contains the machine instructions and assembly code for your program. The line numbers appear as comments, as shown below:

```
SQRT_TEXT       SEGMENT
; Line 6
     PUBLIC     _main
_main     PROC FAR
     *** 000000     55                  push bp
     *** 000001     8b ec               mov  bp,sp
     *** 000003     b8 02 00            mov  ax,2
     *** 000006     9a 00 00 00 00      call __chkstk
     *** 00000b     9b d9 06 00 00      fld  $T20002
     *** 000010     9b d9 1e 02 00      fstp $S14_SIDEA
; Line 7
     *** 000015     9b d9 06 04 00      fld  $T20003
     *** 00001a     9b d9 1e 06 00      fstp $S15_SIDEB
; Line 9
     *** 00001f     9b d9 06 08 00      fld  $T20004
     *** 000024     9a 00 00 00 00      call __FIsqrt
     *** 000029     9b d9 1e 0a 00      fstp $S16_HYP
     *** 00002e     90 9b               fwait
```

Assembly-Listing File

The assembly-listing file contains the assembly code corresponding to your program file, as shown below:

```
SQRT_TEXT       SEGMENT
; Line 6
     PUBLIC     _main
_main     PROC FAR
     push bp
     mov  bp,sp
     mov  ax,2
     call __chkstk
     fld  $T20002
     fstp $S14_SIDEA
; Line 7
     fld  $T20003
     fstp $S15_SIDEB
; Line 9
     fld  $T20004
     call __FIsqrt
     fstp $S16_HYP
     fwait
.
.
.
_main     ENDP
SQRT_TEXT ENDS
END
```

Note that this sample shows the same code as the object listing sample, except the machine instructions are omitted. This ensures the listing is suitable as input for the Microsoft Macro Assembler (MASM).

Combined Source and Object Listing

The combined source and object listing shows one line of your source program followed by the corresponding line (or lines) of machine instructions, as in the following sample:

```
SQRT_TEXT       SEGMENT
;|*** c    This program calculates the length of the hypotenuse of a
;|*** c    right triangle given the lengths of the other two sides.
;|***
;|***      real sidea, sideb, hyp
;|***
;|***      sidea = 3.
; Line 6
     PUBLIC    _main
_main     PROC FAR
      *** 000000     55                 push bp
      *** 000001     8b ec              mov  bp,sp
      *** 000003     b8 02 00           mov  ax,2
      *** 000006     9a 00 00 00 00     call __chkstk
      *** 00000b     9b d9 06 00 00     fld  $T20002
      *** 000010     9b d9 1e 02 00     fstp $S14_SIDEA
;|***      sideb = 4.
; Line 7
      *** 000015     9b d9 06 04 00     fld  $T20003
      *** 00001a     9b d9 1e 06 00     fstp $S15_SIDEB
;|***
;|***      hyp = sqrt(sidea**2 + sideb**2)
; Line 9
      *** 00001f     9b d9 06 08 00     fld  $T20004
      *** 000024     9a 00 00 00 00     call __FIsqrt
      *** 000029     9b d9 1e 0a 00     fstp $S16_HYP
      *** 00002e     90 9b              fwait
;|***
;|***      write(*,100) hyp
.
.
.
_main     ENDP
SQRT_TEXT ENDS
END
;|***
```

Note that this sample is like the object-listing sample, except the program source line is provided in addition to the line number.

Map File

The map file contains a list of segments in order of their appearance within the load module. An example is shown below:

```
Start  Stop   Length Name                   Class
00000H 00059H 0005AH SQRT_TEXT             CODE
0005AH 018E1H 01888H _TEXT                 CODE
.
.
.
```

The information in the `Start` and `Stop` columns shows the 20-bit address (in hexadecimal) of each segment, relative to the beginning of the load module. The load module begins at location zero. The `Length` column gives the length of the segment in bytes. The `Name` column gives the name of the segment, and the `Class` column gives information about the segment type.

The starting address and name of each group appear after the list of segments. A sample group listing is shown below:

```
Origin   Group
0643:0   DGROUP
```

In the example above, `DGROUP` is the name of the data group. `DGROUP` is the only group used by most programs compiled with the Microsoft FORTRAN Compiler, Version 5.0. (Multithread and DLL applications use other groups.)

The map file shown below contains two lists of global symbols: the first list is sorted by symbol address and the second is sorted alphabetically by symbol name. The notation `Abs` appears next to absolute symbol names (symbols containing 16-bit constant values that are not associated with program addresses).

Many of the global symbols that appear in the map file are symbols used internally by the FORTRAN compiler. These symbols usually begin with one or two leading underscores or end with `QQ`.

```
Address         Publics by Name

0005:1594       $i8_output
0005:1855       $i8_tpwr10
0000:FE32  Abs  FIARQQ
0000:0E32  Abs  FICRQQ
0000:5C32  Abs  FIDRQQ
0000:1632  Abs  FIERQQ
0000:0632  Abs  FISRQQ
0000:A23D  Abs  FIWRQQ
0000:4000  Abs  FJARQQ
0000:C000  Abs  FJCRQQ
0000:8000  Abs  FJSRQQ
018E:190B       ICLRER
018E:1932       IGETER
0643:058D       OFF_ARGPTR
```

```
0643:058B       OFF_DESCPT
0643:00F4       STKHQQ
0005:0885       _access
0005:106F       _brkctl
0005:091A       _chsize
.
.
.
0643:00F0       _ _aaltstkovr
0643:0278       _ _abrkp
0643:0228       _ _abrktb
0643:0278       _ _abrktbe
.
.
.
```

Finally, the map file gives the program entry point:

```
Program entry point at 0005:03C9
```

7.19 Searching for Include Files (/I, /X)

Options

/Idirectory [[*/Idirectory...*]]
/X

The /I and /X options temporarily override the environment variable INCLUDE. These options give special handling to a particular file or files, without changing the normal compiler environment.

The /I (include) option causes the compiler to search the directory or directories you specify before it searches the standard places given by the INCLUDE environment variable.

You can search more than one include-file directory by giving the /I option more than once. The directories are searched in order of their appearance on the command line. Each occurrence of an /I option applies only to source files following the option.

The directories are searched only until the include file specified in the source file is found. If the file cannot be found, the compiler prints an error message and stops processing. When this occurs you must restart compilation with a corrected directory specification.

The following list describes the compiler's search order for include files:

1. All the directories in the "parent" file's path. The parent file is the file containing the **INCLUDE** statement or **$INCLUDE** metacommand. For example, if a file named FILE1 includes a file named FILE2, FILE1 is the parent file of FILE2.

Include files can be nested; thus, in the preceding example, FILE2 can include another file named FILE3. In this case, FILE1 is said to be the "grandparent" file of FILE3. For nested include files, the search begins with the directories in the parent file's path, then proceeds through the directories of each of its grandparent files. (See the examples below for an illustration of this procedure.)

2. The directories specified in each /I option.
3. The places specified in the INCLUDE environment variable.

The **INCLUDE** statement or **$INCLUDE** metacommand may give a full or partial path specification for the file. (A full path specification starts with the drive name; a partial path specification gives one or more directory names before the name of the file, but no drive name.) If a full path specification is given for the include file, the compiler uses the given path to find the file, and the INCLUDE environment variable and any /I options have no effect. If a partial path specification is given, the compiler attempts to find that path, starting from the parent file's directory, then from the grandparent file's directories, then from the directories given on the command line, and finally from the directories given by the INCLUDE environment variable.

The /X (exclude) option stops the compiler from searching the standard places given by the INCLUDE environment variable. When /X is given, FL considers the list of standard places to be empty. The parent and grandparent directories are still searched, however.

Like the /I option, /X applies only to source files following the option on the command line. The /X option can be followed by one or more /I options. This causes the compiler to search only the parent and grandparent directories and the directories given by the /I options, ignoring the standard places.

Examples

```
FL /IC:\TESTDIR /IC:\PREVIOUS *.FOR
```

The example above assumes the INCLUDE environment variable is set to `C:\FOR\INCLUDE`. It compiles all source files with the default extension (.FOR) in the current working directory, searching for include files in the following order:

1. The current working directory
2. `\TESTDIR`, the first directory on the command line
3. `\PREVIOUS`, the second directory on the command line
4. `\FOR\INCLUDE`, the directory given by the INCLUDE environment variable

However, if the metacommand $INCLUDE:'\SUB\DEFS' is contained in one of the source files, the compiler adds the subdirectory \SUB to the end of each path it searches. Thus, the search for the include file named DEFS proceeds in the following order:

1. The current working directory (which contains any parent source files)
2. The \SUB subdirectory of the current working directory
3. \TESTDIR, the first directory on the command line
4. \PREVIOUS, the second directory on the command line
5. \FOR\INCLUDE, the directory given by the INCLUDE environment variable

The following example assumes that the INCLUDE environment variable is set to C:\FOR\INCLUDE:

```
FL ..\TESTS\*.FOR
```

It compiles all source files with the default extension (.FOR) in the directory named ..\TESTS, searching directories for include files in the following order:

1. ..\TESTS, the directory containing any possible parent files
2. \FOR\INCLUDE, the directory given by the INCLUDE environment variable

However, if one of the source files in the directory ..\TESTS contains the metacommand $INCLUDE:'\SUB\DEFS', the compiler adds the subdirectory \SUB to the end of each path it searches. Thus, the search for the include file named DEFS proceeds in the following order:

1. ..\TESTS\SUB, adding the subdirectory \SUB to the directory ..\TESTS, where ..\TESTS is the directory containing the parent source file
2. \FOR\INCLUDE\SUB, adding the subdirectory \SUB to the directory \FOR\INCLUDE, where \FOR\INCLUDE is the directory given by the INCLUDE environment variable

If the file \SUB\DEFS contains the metacommand $INCLUDE:'COMS', the compiler searches directories for the nested include file named COMS in the following order:

1. ..\TESTS\SUB, the directory containing DEFS, the parent file of the file named COM
2. ..\TESTS, the directory containing the grandparent source file of the file named COM
3. ..\FOR\INCLUDE, the directory given by the INCLUDE environment variable

In this last case, since COMS is not specified as part of another subdirectory, no subdirectory is added to the end of the path specified in the INCLUDE environment variable.

The search ends when the file is found.

The following example uses a combination of the /I and /X options to control the search path:

```
FL  MAIN.FOR  /X  /ITEST1  SUB1.FOR  /ITEST2  SUB2.FOR
```

Since no /I option appears before MAIN.FOR on the command line, the compiler searches for any files included by MAIN.FOR in the standard places defined by the INCLUDE environment variable (after searching the parent file's directory). Since the /X option precedes the next file name, SUB1.FOR, the compiler does not search the standard places for any files SUB1.FOR includes (in this case, the environment variable is not used). Instead, only the directory of the parent source file SUB2.FOR and the directory TEST1 are searched. If the include file or files cannot be found in one of those places, an error occurs. The second /I option adds one more directory to be searched for any include files specified in the parent file SUB2.FOR. The TEST2 subdirectory is searched after the TEST1 subdirectory.

7.20 Handling Warnings and Errors

There are several kinds of errors that can occur when a program is compiled, linked, and run. Section 7.20.1 gives an overview of Microsoft FORTRAN error messages. Several options are available to control the types of warnings generated at compile time and to enable or disable expanded error handling at run time. See Section 7.20.2 for a description of these options.

7.20.1 Understanding Error Messages

Errors can occur at any stage of program development, as explained below:

1. During compilation, the compiler generates a broad range of error and warning messages to help you locate errors and potential problems in your source files.
2. While linking, the linker is responsible for generating error messages.
3. When a program is executed, any error messages are run-time error messages. This category includes messages about floating-point exceptions, which are errors generated by an 8087, 80287, or 80387 coprocessor.

Other utilities included in this package, such as NMAKE and EXEMOD, generate their own error messages. You can also distinguish the type of a message by its format. See Appendix D, "Error Messages," for a description of error-message formats, a list of actual error messages, and explanations of the circumstances that cause them.

When you are compiling and linking using the FL command, you may see both compiler and linker messages. The LINK program banner appears on the screen when linking begins. Compiler messages are any messages that appear before the LINK banner, and linker messages are those that appear after the banner. Compiler messages have numbers preceded by the letter `F`, and linker messages have numbers preceded by the letter `L`.

Compiler error messages are sent to standard output, which is usually your terminal. You can redirect the messages to a file or printer by using one of the DOS redirection symbols: **>** or **>>**.

Error redirection is useful in batch-file processing. For example, the following command redirects error messages to the printer device (designated by **PRN**):

```
FL  /c  COUNT.FOR > PRN
```

See Section 7.15, "Special File Names," or your DOS documentation for a list of device names, including **PRN**.

In the following command, only output that ordinarily goes to the console screen is redirected.

```
FL  COUNT.FOR > COUNT.ERR
```

The FL control program returns an exit code indicating the compilation status. Exit codes are useful with the DOS batch command **IF ERRORLEVEL** and with the NMAKE utility. You can use them to test for the success or failure of the compilation before proceeding with other tasks. See Appendix B, "Using Exit Codes," in the *Microsoft CodeView and Utilities User's Guide* for more information.

7.20.2 The Warning-Level Option (/W)

Option

/W{0 | 1| 2}

You can suppress compiler warning messages with the /W (warning) option. Any message beginning with `F4` is a compiler warning message. Warnings indicate potential problems (rather than actual errors) with statements that may not be compiled as you intend.

/W1 (the default) causes the compiler to display warning messages. /W0 turns off warning messages. The /W0 option is useful when compiling programs that deliberately include questionable statements. /W2 suppresses the following error messages:

F4998 variable used but not declared

F4999 variable declared but not used

/W0 and /W2 apply to the remainder of the command line or until the next occurrence of /W1 on the command line. These options have no effect on object files given on the command line.

Example

```
FL  /W0  CRUNCH.FOR  PRINT.FOR
```

This example suppresses warning messages when the files CRUNCH.FOR and PRINT.FOR are compiled.

7.21 Syntax Errors (/Zs)

Option

/Zs

The /Zs option tells the compiler to perform a syntax check only. This is a quick way to find and correct syntax errors before compiling a source file. With /Zs, no code is generated and no object files or object listings are produced. However, you can specify the /Fs option on the same command line to generate a source listing.

The /Zs option applies to all source files that follow the option on the command line but does not affect any source files preceding the option.

Example

```
FL  /Zs  TEST*.FOR
```

This command causes the compiler to perform a syntax check on all source files in the current working directory that begin with TEST and end with the default extension (.FOR). The compiler displays messages for any errors found.

NOTE *The /4Yb and /4Yd options discussed in Section 7.9.2 above are also useful in identifying errors.*

7.22 Preparing for Debugging (/Zi, /Od, /Zd)

Option	Effect
/Zi	Prepares for debugging with the Microsoft CodeView debugger
/Od	Disables optimization
/Zd	Prepares for debugging with SYMDEB

The /Zi option produces an object file containing full symbolic debugging information—including the symbol table and line numbers—for use with the Microsoft CodeView window-oriented debugger.

When you use the FL command to compile and link, giving the /Zi option automatically causes the /CO option to be given at link time. If you link separately (whether using FL or the LINK command), instead of compiling and linking in one step, be sure to give the /CO option when you link. Otherwise, symbols and source-code lines will be missing when you run the CodeView debugger. See Section 1.3.5, "Preparing FORTRAN Programs," in the *Microsoft CodeView and Utilities User's Guide* for more information on /CO.

The /Od option tells the compiler not to optimize. The default is to optimize. Using /Od is recommended whenever you use /Zi. It is also recommended while testing, since it can improve compilation speed by 30 to 35 percent.

NOTE *If you use /Od when compiling, the F3S.EXE file must be in the current search path.*

Since optimization may rearrange instructions and store values in machine registers, you may have trouble finding and fixing errors if you optimize before debugging.

Note that turning off or restricting optimization of a program usually increases the size of the generated code. If your program contains a module that is close to the 64K limit on compiled code, turning off optimization may cause the module to exceed the limit.

See Section 7.24, "Optimizing," for a discussion of additional optimization options.

NOTE *When the debug option (/4Yb) is enabled, loop optimization is disabled. See Section 7.9.2.1 for a description of the debug option.*

The /Zd option produces an object file containing line-number records that correspond to the line numbers of the source file. The /Zd option is used when you want to pass an object file to the SYMDEB symbolic debugger, available with other Microsoft products (for instance, FORTRAN Version 3.0 and earlier). The debugger can use the line numbers to refer to program locations. However, only global symbol-table information is available with SYMDEB (unlike the CodeView debugger, which also recognizes local symbols).

When you use the FL command to compile and link, giving the /Zd option causes the /LI option to be given at link time. (See Chapter 20, "Linking Object Files with LINK," in the *Environment and Tools* for more information on /LI.) If you compile a source file with the /Zd option, and then link in a separate step using FL, be sure to give the /Zd option when you link. (If you link using the LINK command, give the /LI option.) Otherwise, your executable file will not contain line numbers.

The /Zd option generates a map file, whether or not the /Fm option is given. If /Fm is not used to specify a file name or location for the file, the map file is created in the current working directory and given the default name, as described in Section 7.14.

The /Zi, /Od, and /Zd options apply to any source files following the option on the command line, but do not affect source files preceding the option. The /Zi and /Od options have no effect on object files given on the command line. /Zd causes the /LI option to be given at link time.

Example

```
FL  /Zi  /Od  /Fs  P*.FOR  /FePROCESS  /FmPROCESS
```

This command compiles all source files in the current working directory beginning with `P` and ending with the default extension (.FOR), creating object files that contain the symbolic information needed by the CodeView debugger. Optimization is disabled with /Od. The /Fs option creates a source listing for each source file. The executable file is named PROCESS.EXE, and a map file named PROCESS.MAP is also created.

7.23 Using an 80186, 80188, 80286, or 80386 Processor (/G0, /G1, /G2)

Option	Effect
/G0	8086/8088 instruction set (default)
/G1	80186/80188 instruction set
/G2	80286/80386 instruction set

If you have an 80186/80188 or 80286/80386 processor, you can use the /G1 or /G2 option to enable the instruction set for your processor. Use /G1 for the 80186 or 80188 processor; use /G2 for the 80286 or 80386. (80286 code runs on the 80386, but does not use any of the 80386's specialized instructions.) Although it is advantageous to use the appropriate instruction set, you are not required to do so. If you have an 80286 processor, for example, but want your code to also run on an 8086, do not use the /G1 or /G2 option.

The /G0 option enables the instruction set for the 8086/8088 processor. You do not have to specify this option explicitly since the 8086/8088 instruction set is used by default. Programs compiled this way also run on an 80186, 80188, 80286, or 80386 processor.

Only one of these three options is allowed on the FL command line. If more than one appears, FL issues a warning and generates code using the last /G option on the line.

Example

```
FL  /G2  /FeFINAL  *.FOR
```

The example above compiles and links all source files with the default extension (.FOR) in the current working directory, using the 80286 instruction set. The resulting program, named FINAL.EXE, will run only on an 80286 or 80386.

7.24 Optimizing (/O and /Zp)

The optimizing procedures performed by the Microsoft FORTRAN Compiler can reduce the storage space and execution time required for a compiled program by eliminating unnecessary instructions and rearranging code. The compiler performs some optimizations by default. You can use the /O options to exercise greater control over the optimization performed.

Option

/Oletters

The /O (optimize) option controls optimization. The *letters* after /O alter the way the compiler optimizes your code. The *letters* are one or more of the following:

Character	Optimizing Procedure
d	Disables optimization; leaves stack checking on
l	Loop optimization
p	Improves consistency of floating-point results
s	Favors reduced code size
t	Favors rapid execution
x	Full optimization; equivalent to /Olt /Gs

More than one /O option may appear on a command line. All of the options apply to all of the source-code files. FL processes the options in the order in which they appear, so their order is significant. For example, /Od followed by /Ol (or only /Odl) disables all optimization, then reinstates loop optimization only. On the other hand, /Ol followed by /Od (or just /Old) requests loop optimization, then disables all optimization. Hence, no optimization of any kind is performed.

When you omit the /O option, or when you give an /O option but do not use the letter x, the compiler defaults to /Ox. Whenever the compiler has a choice between producing smaller (but slower) code and larger (but faster) code, the compiler chooses to generate the larger, faster code. To make the compiler produce smaller code, use the /Os option.

The /Od option turns off optimization. This option is useful in the early stages of program development because it avoids optimizing code that is not in its final form, and it improves compilation speed by approximately 30 to 35 percent. Because optimization may involve rearrangement of instructions, you may also want to specify the /Od option when you use a debugger other than the CodeView debugger with your program, or when you want to examine an object listing. (The /Zi option, which prepares a program for debugging with the CodeView debugger, automatically turns off loop optimization and optimization involving code rearrangement.) If you optimize before debugging, it can be difficult to recognize and correct your code.

You may add options to the /Od option to reinstate specific optimizations. For example, /Odl performs only loop optimization, and /Odtp favors speed and floating-point consistency without optimizing loops. Any consistent combination

is permitted. For example, /Odts would not be allowed since you cannot simultaneously optimize for size and speed.

Note that turning off or restricting optimization usually increases the program's size. If your program contains a module that is close to the 64K limit on compiled code, turning off optimization may cause the module to exceed the limit.

NOTE *In performing optimizations on extremely complex code, the compiler may experience an internal error. It is sometimes possible to work around this problem by disabling the optimization pass with the /Od option.*

In all cases where you experience this type of compiler error, please contact Microsoft Corporation so that corrections can be made for subsequent releases.

The /Op option is useful when floating-point results must be consistent. This option changes the compiler's default handling of floating-point values. Normally, the compiler tries to avoid assigning any value to a variable until all the calculations required for the variable's final value have been completed. It does this by storing intermediate values (wherever possible) in an 80-bit machine register.

However, since floating-point types are allocated less than 80 bits of storage (32 bits for **REAL*4** and 64 bits for **REAL*8**), a register value may actually be more precise than the same value stored in a floating-point variable. Over the course of many calculations, the value that results from the use of a machine register may be different from the value produced if the compiler assigned intermediate results to a variable. Furthermore, adding or deleting code may change the number of machine registers available to hold intermediate results, and thus alter the result of a particular calculation.

Specifying the /Op option tells the compiler to place all intermediate results in variables rather than machine registers. Although this may give less precise results than using registers, and may increase program size, it guarantees consistent results in floating-point calculations.

NOTE *When the debug option (/4Yb) is enabled, loop optimization is disabled. See Section 7.9.2.1 for a description of the debug option.*

Examples

```
FL  /c  /Os  FILE.FOR
```

The command above favors code size over execution speed when compiling `FILE.FOR`.

```
FL  /Od  *.FOR
```

The command above compiles and links all FORTRAN source files with the default extension (.FOR) in the current directory and disables optimization. This

command is useful during the early stages of program development, since it improves compilation speed.

```
FL  /Op  /FeTESTRUN  *.FOR
```

The command above causes floating-point assignments to variables to be carried out immediately (where specified) when compiling all source files with the default extension (.FOR) in the current working directory. By default, the optimization favors execution time. The resulting program is named TESTRUN.EXE.

Option

/ Zp[[{1 | 2 | 4}]]

The / Zp option controls the starting addresses (packing) of variables in structures. A tight pack saves memory space, at the expense of slightly slower access for noncharacter variables. Any **$PACK:***n* metacommand in the file overrides a / Zp option, except for **$PACK**, which restores whatever command-line option was given.

If no / Zp option is given, and if there is no **$PACK** metacommand in the file, structures are packed according to the following default rules: **INTEGER*1**, **LOGICAL*1**, and all **CHARACTER** variables begin at the next available byte, whether odd or even; all other variables begin at the next even byte. This arrangement wastes some memory space, but gives the quickest access.

If / Zp1 is specified, all variables begin at the next available byte, whether odd or even. Although this slightly increases access time, no memory space is wasted.

If / Zp2 is specified, **INTEGER*1**, **LOGICAL*1**, and all **CHARACTER** variables begin at the next available byte, whether odd or even. All other variables start on the next available even byte. This is equivalent to the default packing, described above.

If / Zp4 is specified, **INTEGER*1**, **LOGICAL*1**, and all **CHARACTER** variables begin at the next available byte, whether odd or even. All other variables start on the next four-byte boundary.

7.25 Enabling and Disabling Stack Probes (/Ge, /Gs)

Option	Effect
/Ge	Enables stack probes
/Gs	Disables stack probes (default)

A stack probe is a short subroutine called on entry to a procedure to verify that the program stack has enough space for any automatic local variables. When stack probes are enabled, the stack-probe subroutine is automatically called at

every entry point. The stack-probe subroutine generates a message and ends the program if it determines that the required stack space is not available.

By default, all Microsoft FORTRAN variables are static, not automatic, so only procedures containing automatic variables require stack space for variable allocation. However, any procedure call consumes stack space for the return address and the addresses or values of the arguments passed. If your program has many nested procedure calls, or passes a lot of variables to procedures, the stack may overflow. Although enabling a stack probe does not prevent an overflow, it will tell you when one occurs.

Note, however, that programs with stack checking are slightly larger and may perform less efficiently due to the calls to the probe routine. Stack probes can be used during development, then removed from the final version of the program.

The /Ge option applies to all source files following the option on the command line. The /Gs option disables stack checking for all source files that follow it on the command line.

NOTE *Although the default option, which disables stack probes, reduces program size, it means that no compiler error message is displayed if a stack overflow occurs. You may want to use the /Ge option when testing to make sure the program does not cause a stack overflow.*

Example

```
FL  /c  /Ge  /Ot  FILE.FOR
```

This example enables stack probes and favors execution time when compiling FILE.FOR.

7.26 Suppressing Automatic Library Selection (/Zl)

Option

/ Zl

The compiler ordinarily places in the object file the name of the FORTRAN library corresponding to the floating-point and memory-model options you choose. The linker uses the library name to link the program automatically with the corresponding library. Thus, you do not need to specify a library name to the linker, provided that the appropriate library exists for the floating-point and memory-model options you are using.

The / Zl option suppresses the insertion of library names in object files. When you specify / Zl, the compiler does not place a library name in the object file. As a result, the object file is slightly smaller.

The / Zl option produces a significant space savings when building a library containing many object modules. When you link a library created using the / Zl

option with a program file compiled without the / Zl option, the program file supplies the name of the library.

The / Zl option applies to the remainder of the source files on the command line.

Examples

```
FL  ONE.FOR  /Zl  TWO.FOR
```

The example above creates an object file named ONE.OBJ. Since no floating-point or memory-model options are specified on the FL command line, this object file contains the name of the FORTRAN library that corresponds to the default floating-point and memory-model options (LLIBFOR7.LIB). The example also creates an object file named TWO.OBJ without any library information, since the / Zl option appears before the file name on the command line. When ONE.OBJ and TWO.OBJ are linked to create an executable file, the library information in ONE.OBJ causes LLIBFOR7.LIB to be searched for any unresolved references in either ONE.OBJ or TWO.OBJ.

```
FL  /c  /Zl  *.FOR
```

The example above compiles all source files with the default extension (.FOR) in the current working directory. None of the resulting object files contain library information.

7.27 Setting the Stack Size (/F)

Option

/F *hexnum*

The /F option sets the size of the program stack. A space must separate /F and *hexnum*.

The *hexnum* is a hexadecimal number representing the stack size in bytes. The number must be positive and cannot exceed 10,000 hexadecimal (65,536 decimal). The default stack size is 2K.

If many variables are declared as automatic, or if many parameters are passed to procedures, a larger stack may be needed to accommodate them. Stack overflow does not always cause a program to crash; it may have no effect at all, or it may cause erratic operation or incorrect results. If there is any doubt whether the stack is large enough, use the /Ge option to enable stack probes. Stack overflow will then automatically halt the program.

NOTE *By default, stack checking is disabled (see Section 7.25). When stack checking is disabled, problems caused by stack overflow can be difficult to diagnose. If you suspect such problems, enable stack probes until your program is debugged.*

Using the /F option with the FL command has the same effect as using the /STACK option with the LINK program.

Example

```
FL /F C00 *.OBJ
```

This example sets the stack size to C00 hexadecimal (3K decimal) for the program created by linking all object files in the current working directory.

7.28 Restricting the Length of External Names (/H)

Option

/H*number*

The /H option restricts the length of external names in the object file. The *number* is an integer specifying the maximum number of significant characters in external names.

The /H option has no effect during compilation. The first 31 characters of a name are significant if **$NOTRUNCATE** (the default) is in effect, and only the first six if **$TRUNCATE** is in effect. It is only when the object file is created that any external names longer than *number* characters are truncated.

This option has no effect on local names.

7.29 Labeling the Object File (/V)

Option

/V"*string*"

Use the /V (version) option to embed a given text *string* into an object file. The quotation marks may be omitted if the string does not contain blanks or tabs. A common use of the /V option is to label an object file with a version number or copyright notice.

The /V option applies to all source files following the option on the command line.

Example

```
FL /V"Microsoft FORTRAN Compiler Version 5.0" MAIN.FOR
```

This command places the string `Microsoft FORTRAN Compiler Version 5.0` in the object file MAIN.OBJ.

7.30 Linking with Libraries

When the FL command compiles a source file, it places a FORTRAN library name in the object file. The library name corresponds to the memory-model and floating-point options you chose on the FL command line, or the defaults for those options you did not explicitly select. See Table 7.1 in Section 7.4, "Memory-Model Options," for the library names FL includes in the object file for each combination of memory-model and floating-point options.

The linker looks for a library matching the name embedded in the object file. If it finds a library with that name, it automatically links the library with the object file.

The result is that you do not need to specify library names on the FL command line unless you want to link with standard libraries created during SETUP, with libraries other than the default library for the floating-point and memory-model options you have chosen, or with user-created libraries.

Setting up for both real- and protected-mode programming creates separate real- and protected-mode libraries, with different names. Use the /L option to specify the correct link library. See Section 7.5, "Library Options," for more information.

If you want to link with other libraries, you must either use the /link option on the FL command line and include the new library names, or run the linker and specify the library names separately. In either case, the linker tries to resolve external references by searching the library you specified before it searches the library whose name is embedded in the object file. If you want the linker to ignore the library whose name is embedded in the object file, you must also include the /NOD (NODEFAULTLIBRARYSEARCH) linker option, either as part of the /link option on the FL command line, or as an option on the LINK command line.

See Section 7.32, "Using FL to Link without Compiling," for information about the /link option of the FL command. See Chapter 20, "Linking Object Files with LINK," in the *Environment and Tools* for information about specifying library names to the linker.

See Section 7.4, "Memory-Model Options," for more information on dynamic-link library and multithread linking.

7.31 Creating Overlays

You can specify program overlays on the command line. Overlays let several program modules use the same memory area, one module at a time. When needed, a module is loaded from the disk. See Chapter 20, "Linking Object Files with LINK," in the *Environment and Tools* for more information about overlays.

The modules to be overlaid are enclosed in parentheses. For example, the following command line instructs FL to overlay modules `OVER1.FOR` and `OVER2.FOR`:

```
FL  MAIN.FOR  (OVER1.FOR  OVER2.FOR)
```

7.32 Using FL to Link without Compiling

Just as the FL command can compile files without linking the resulting object code, you can use FL to link object files that were previously compiled. If all of the files you give FL have extensions other than .FOR, and if no /Tf options appear, FL skips the compiling stage and links your files. To link object files, use the following special form of the FL command:

FL *objfile*[[,*objfile*...]] /link [[*libfield*]] [[*linkoptions*]]

When FL links object files, it gives the resulting executable file the base name of the first object file on the command line, plus an .EXE extension, by default. Alternately, you can specify files with the .LTB extension without the /link option. The /link option is most useful for passing link options. (This is the same naming convention FL uses when it compiles source files first, then links the resulting object files.)

Command options beginning with /F allow you to supply the file names and options that would otherwise appear on the LINK command line (or in response to LINK prompts). The following list shows each FL option for the linker and the corresponding LINK command-line field, prompt, or option:

FL Option	LINK Field/Prompt/Option
/Fe*exefile*	The *exefile* field or "Run File" prompt
/Fm*mapfile*	The *mapfile* field or "List File" prompt
/link *libfile linkoptions*	The *libfiles* field or "Libraries" prompt, and any of the LINK options described in Chapter 13 of the *Microsoft CodeView and Utilities User's Guide*
/F*hexnum*	The /STACK option

See Section 7.13 for a description of the /Fe option, Section 7.14 for a description of the /Fm option, and Section 7.27 for a description of the /F option. Chapter 20 of the *Environment and Tools* describes the LINK command-line fields and the /STACK option.

If you use the /link *libfile linkoptions* option with the FL command, it must be the last option on the command line.

NOTE *The FL command normally links without having to specify the /link option. However, you can include the /link option in an FL command line if you wish to modify the linking process, such as by specifying a different library.*

7.33 Specifying Assembler Options (/MA)

Option

/MA *option*

The /MA option allows you to specify Microsoft Assembler options when including assembly-language filenames on the command line. You must use a separate /MA option for each assembly-language option. The option applies to all of the assembly-language files that follow it on the command line, unless another option overrides it.

Example

```
FL FORT1.FOR /MA/B40 ASM1.ASM
```

In this example, the /B40 option is passed to the Microsoft Assembler, specifying that the source file buffer is to be 40K when ASM1.ASM is assembled.

7.34 Generating a Source Browser Database (/Fr)

Option

/Fr [[*browsefile*]]

The /Fr option generates a standard PWB Source Browser database for the source file being compiled by the FL command. The resulting file is for use with PWBRMAKE.EXE.

The optional *browsefile* parameter specifies the name for the generated database file. If *browsefile* is omitted, FL gives the file the same base name as the source file, plus the extension .SBR.

The /Fr option may appear anywhere on the command line prior to any /LINK options. However, the database files will be generated only for those source files that follow /Fr on the command line, not those that appear before it.

Example

```
FL /Fr DATAB.SBR MAIN.FOR
```

7.35 Generating an Extended Source Browser Database (/FR)

Option

/FR [[*browsefile*]]

The /FR option generates an extended PWB Source Browser database for the source file being compiled by the FL command. This includes information about parameters and local variables, as well as definitions and references for global variables, subroutines, and functions.The resulting file is for use with PWBRMAKE.EXE.

The optional *browsefile* parameter specifies the name for the generated database file. If *browsefile* is omitted, FL gives the file the same base name as the source file, plus the extension .SBR.

The /FR option may appear anywhere on the command line prior to any /LINK options. However, the database files will be generated only for those source files that follow /FR on the command line, not those that appear before it.

Example

```
FL /FR EXTDATAB.SBR MAIN.FOR
```

7.36 Setting IBM VS Compatibility (/4Yv)

Option

/4{ Y | N }V

This option enables (Y) only those extensions that are compatible with the IBM VS FORTRAN compiler, or allows (N) all Microsoft extensions. /4NV is the default.

Example

```
FL /4YV MAIN.FOR
```

For a complete list of Microsoft FORTRAN language extensions, see section 7.9.4, "Controlling Optional Language Features (/4Ys, /4Yi, /4Yv)."

7.37 Pascal Convention for Passing Parameters (/Gb)

Option

/Gb

By default, Microsoft FORTRAN 5.1 passes procedure parameters by address in a manner compatible with C. Previous versions, however, passed procedure parameters using the Pascal convention (by description).

The /Gb option forces parameters to pass by the Pascal method. The option is included for backwards compatibility with object modules produced by previous versions of the Microsoft FORTRAN compiler, and for compatibility with Pascal.

Example

```
FL /Gb MAIN.FOR
```

7.38 Creating a QuickWin Application (/MW)

Option

/MW [[*number*]]

The /MW option instructs FL to compile a QuickWin application. QuickWin applications run under Microsoft Windows version 3.0 or later, in a character-mode window that mimics the DOS environment.

QuickWin applications are not restricted to the DOS 640K limit; they may access all of the memory available under Windows 3.0.

NOTE *Windows must be running in either Standard or 386 Enhanced mode to run a QuickWin application.*

The *number* in the /MW option specifies where FL inserts calls to the **YIELDQQ** subroutine. If *number* is not specified, /MW defaults to /MW1. (For more information on **YIELDQQ**, see Chapter 7 in *Advanced Topics*.) The value of *number* has different effects:

Value	Description
0	Leaves the program unchanged. You must add any **YIELDQQ** statements manually.
1	Inserts **YIELDQQ** calls before **DO** statements.
2	Inserts **YIELDQQ** calls before each entry to a user-written routine.
4	Inserts **YIELDQQ** calls before **READ** and **WRITE** statements.

You can add the numerical values of the options to combine their effects. For example, /MW3 inserts **YIELDQQ** statements before **DO** statements and before each call to a user-written routine.

All QuickWin applications are automatically linked with the standard module-definitions file FL.DEF (stored in the same directory as FL.EXE) and the library LLIBFEW.LIB. You do not need to include either of them on the comand line.

NOTE *QuickWin applications are always large model.*

Example

```
FL /MW MRTX_WIN.FOR
```

This command inserts **YIELDQQ** statements before each **DO** statement in MTRX_WIN.FOR; compiles MTRX_WIN.FOR; links the resulting object file to the module-definition file FL.DEF and to the QuickWin library LLIBFEW.LIB; and produces the Windows executable file MTRX_WIN.EXE.

You can start QuickWin programs either from the command line:

```
WIN MTRX_WIN
```

or from within Windows using the Program Manager.

Appendixes

Appendix A

ASCII Character Codes

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂†

† ASCII code 127 has the code DEL. Under DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL + BKSP key combination.

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	░
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	█
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

Appendix B

Differences from Previous Versions

This appendix documents the changes to Microsoft FORTRAN as the language has evolved from Versions 3.2 and 3.3 to Versions 4.0, 4.1, 5.0, and 5.1. The appendix is in four sections, one for each update.

The first section describes the current update, from Version 5.0 to Version 5.1. The second section describes changes from Version 4.1 to Version 5.0. The third section explains the changes that occurred when OS/2 features were added to Version 4.0 to create Version 4.1. The fourth section describes the changes from Versions 3.2 and 3.3 to Version 4.0.

The README.DOC file may also contain information unavailable when this update was printed.

You may use the SETUP program to create libraries to link with Version 5.1 programs. If you choose all the default responses for SETUP, the library that SETUP builds requires that you have an 8087 or 80287 coprocessor installed.

NOTE *Since MS-DOS and PC-DOS are functionally the same operating system, Microsoft manuals and updates use the term DOS to include both systems, except when noting features that are unique to one or the other.*

B.1 Changes from Version 5.0 to Version 5.1

The principal change from Version 5.0 to Version 5.1 is the addition of Windows 3.0 support. The compiler, the linker, and the FL utility have been modified to correctly compile and link Windows 3.0 programs.

B.1.1 Windows 3.0 Support

You can rebuild most FORTRAN programs written for DOS or OS/2 to run under Windows 3.0 without any changes. The QuickWin run-time libraries create a window for the program and treat it like a terminal. Your application can write to and read from this window with standard FORTRAN input and output routines. For detailed information, see Chapter 7, "Programming for Windows," in Microsoft FORTRAN *Advanced Topics*.

B.1.2 Extended FL Utility

The new version of FL includes options for such actions as compiling Windows programs and IBM VS compatible programs, and adding Source Browser information to object files. You can also obtain online help for FL.

B.1.2.1 Compiling for Windows 3.0

You can use the following options for compiling Windows programs:

Option	Description
/Aw	Compile for Windows DLL
/Gw	Compile for Windows DLL entry points
/MW	Compile for Windows EXE

For detailed information, see Chapter 7, "Programming with Windows," in Microsoft FORTRAN Advanced Topics.

B.1.2.2 Backward Compatibility for Passing Parameters

In FORTRAN 5.1, actual arguments which are external functions or subroutines are passed by address rather than by descriptor. (Passing by address is compatible with C; passing by descriptor is the Pascal method.) The /Gb option will compile programs using the parameter passing method from FORTRAN 5.0 and Pascal. For more detailed information, see the FORTRAN *Quick Reference Guide*.

B.1.2.3 IBM VS Compatibility

The /4{Y|N}V option will compile programs to be compatible with IBM VS FORTRAN.

B.1.2.4 Source Browser Information

You can include Source Browser information in your object files. The /Fr option adds basic Source Browser information and the /FR option adds extended Source Browser information.

B.1.2.5 Online Help for FL

The /help option invokes the QuickHelp utility and displays online help for FL.EXE. FL /? now displays a list of compiler options, called FL.MSG.

B.1.2.6 Suppressing the Compiler Logo

You can suppress the compiler logo by using the /nologo option.

B.1.2.7 Ignoring Case for Routines

The /Zc option ignores case for routines declared with the PASCAL attribute.

B.1.3 BYTE Data Type

The **BYTE** data type is new in Version 5.1. Defining an item as **BYTE** is the same as defining it as **INTEGER*1**.

B.1.4 New Microsoft FORTRAN Functions and Subroutines

See Chapter 5, "Intrinsic Functions and Additional Procedures," for full descriptions of the following functions and subroutines.

Function or Subroutine	Description
INTDOSQQ	Invokes the DOS system call
INTDOSXQQ	Invokes the DOS system call using segment-register values
MATHERRQQ	Processes errors generated by the math library
RAISEQQ	Sends a signal to the executing program
SIGNALQQ	Enables interrupt handling from the operating system

B.1.5 Heap Management for Mixed-Language Programming

Under FORTRAN 5.0 and Microsoft C 5.1, calls to **_fmalloc were** mapped to **_nmalloc** if there was no free far heap. This mapping is not performed under FORTRAN 5.1 and C 6.0. You may encounter the error R6009 or F6700 when using FORTRAN 5.1 and C 6.0 when there is no far heap left, even if there is near heap remaining. This is especially likely if your application's load size is within 64K of the maximum available memory. To fix this problem, you need to write your own version of **malloc( )** which will first allocate from the far heap and then try the near heap if there is no far heap. You'll also need to write versions of **realloc( ) and free( )** which can determine if the argument is far or near and call the appropriate function. For further information, consult your Microsoft C documentation.

B.2 Changes from Version 4.1 to Version 5.0

This section summarizes the Microsoft FORTRAN features introduced in Version 5.0.

B.2.1 Alphabetical Summary

Changes to individual statements, metacommands, and other features are listed alphabetically below.

B.2.1.1 Allocatable Arrays

Arrays may be declared as allocatable. Although the number of dimensions is fixed during compilation, the size and range of each dimension can be set at run time.

B.2.1.2 APPEND Mode

Files may be opened in **ACCESS='APPEND'** mode, which automatically positions the file pointer at the end of the file.

B.2.1.3 Array Operations

A complete array can now be an operand in an expression. For example, two arrays may be directly added together in a single statement, without having to use a **DO** loop. Similarly, a constant value may be subtracted from every element in an array, or two arrays of **LOGICAL** variables can be compared with a logical operator.

An array may also be passed to a function (intrinsic or external). The function returns an identically-dimensioned array whose individual elements are the values obtained when the function is applied to the corresponding element of the original array. For more information, see Section 1.7.5, "Array Expressions."

B.2.1.4 AUTOMATIC Variables

Local variables may be declared "on the stack," rather than at fixed memory locations, with the **AUTOMATIC** statement.

B.2.1.5 Compatibility with Versions 3.2 and 3.3

Version 5.0 no longer supports compatibility with object files produced by Version 3.2 and 3.3 compilers. If you wish to use their code in new programs, they must be recompiled with the Version 5.0 compiler.

B.2.1.6 Conditional Compilation

The **$DEFINE**, **$UNDEFINE**, **$IF**, **$ELSE**, **$ELSEIF**, and **$ENDIF** metacommands control conditional compilation of FORTRAN source code. FL command-line arguments permit test values to be passed to these metacommands.

B.2.1.7 CYCLE Statement

The **CYCLE** statement transfers control back to the first line of a **DO** or **DO WHILE** loop.

B.2.1.8 DO WHILE Statement

The new **DO WHILE** statement permits a loop to be executed as long as a logical condition is **.TRUE.**.

B.2.1.9 END DO Statement

The **END DO** statement can now terminate a **DO** or **DO WHILE** loop. Labeled terminating statements are no longer required (though they are still permitted).

B.2.1.10 End-of-line Comments

A comment line may now begin with an exclamation point. An exclamation point outside a character or Hollerith constant that is not in column six is considered the beginning of a comment, and it may appear anywhere on a line following the statement.

B.2.1.11 Exclusive Or (.xor.) Logical Operator

The **.XOR.** logical operator has been added. Although its function is identical to the existing **.NEQV.** operator, its meaning is clearer.

B.2.1.12 EXIT Statement

The **EXIT** statement transfers control to the first statement following a **DO** or **DO WHILE** loop.

B.2.1.13 IMPLICIT NONE Statement

The new **IMPLICIT NONE** statement removes implicit data typing. Any variable not explicitly typed produces a compile-time warning message.

B.2.1.14 INCLUDE Statement

The **INCLUDE** statement inserts the contents of a specified file at the point in the program where the **INCLUDE** statement appears. The compiler immediately processes these statements. The **$INCLUDE** metacommand, which has the same function, is retained.

B.2.1.15 Loop Optimization

Loop optimization may be enabled and disabled with the **$LOOPOPT** and **$NOLOOPOPT** metacommands.

B.2.1.16 NAMELIST Statement

The **NAMELIST** statement defines a group of variables that are written to or read from a formatted file with a single I/O statement. When written, the namelist variable values are labeled with their names. When read, a variable is assigned the value given in a *varname* = *value* statement. The namelist feature allows a programmer to display a large group of variables with only a single statement, or to read the values of a group of parameters from an ASCII file.

B.2.1.17 Numeric Arguments for Logical Operators

The **.AND.**, **.OR.**, **.XOR.**, **.EQV.**, **.NEQV.**, and **.NOT.** logical operators now accept **INTEGER** arguments. The result of the operation is not a logical value, but an integer value determined by bitwise comparison of the operands.

B.2.1.18 Quotes

Character literals may now be delimited by quotes, as well as apostrophes.

B.2.1.19 SELECT CASE Statement

The **SELECT CASE** construct is similar to other languages' case-selection mechanisms. The test expression may be of **INTEGER**, **LOGICAL**, or **CHARACTER*1** type. The case-list values may be single values, ranges of values, or any combination of the two. An optional **CASE DEFAULT** statement handles situations when none of the list values match the test expression.

B.2.1.20 Structure Type

A user-defined data type called a "structure" may be created. A structure type is a combination of other data types, including other structures. The **RECORD** statement creates variables of a specific structure type. Structure variables may be used both for internal data representation, and for reading and writing groups of data from unformatted files.

B.2.1.21 Symbol Table Enlargement

The Microsoft FORTRAN compiler has changed its internal allocation of the symbol table from near to far memory, allowing many more symbols in a given subprogram before the compiler runs out of memory.

B.2.1.22 $TRUNCATE

Microsoft FORTRAN no longer sets **$TRUNCATE** by default. Programs which contain variable names longer than six characters but with differences after the sixth character will need to explicitly set the **$TRUNCATE** metacommand.

B.2.2 New Microsoft FORTRAN Functions and Procedures

See Chapter 5 , "Intrinsic Functions and Additional Procedures," for full descriptions of the following functions and procedures.

Function or Procedure	Description
EPSILON	Smallest number that is larger than one
GETARG	Returns *n*th command-line argument
HUGE	Largest positive number
LEN_TRIM	Returns the length of a string (less trailing blanks)
MAXEXPONENT	Largest positive exponent for a data type
MINEXPONENT	Largest negative exponent for a data type
NARGS	Returns number of command-line arguments
NEAREST	Closest number (larger or smaller) to a given number
SCAN	Searches a string for a character in a set
TINY	Smallest positive number
VERIFY	Searches a string for a character not in a set

B.2.3 Microsoft FORTRAN Language Extensions

Microsoft FORTRAN Version 5.0 supports all IBM SAA extensions. Microsoft FORTRAN Version 5.0 includes many (but not all) VAX extensions. Some of these features may be implemented in this manual differently than DEC implementation. Read the appropriate section to see if there are differences that are important to your application.

For ease in transporting SAA or VAX applications to MS-DOS computers, there are two FL command-line options which disable all Microsoft FORTRAN extensions except for SAA extensions (/4Yi) or VAX extensions (/4Yv). See Chapter 7, "The FL Command," for more information on these options.

B.2.4 OS/2

Microsoft FORTRAN now supports a method for building dynamic-link and multithread libraries. For more information, see *Microsoft FORTRAN Advanced Topics.*

B.2.5 Graphics

Microsoft FORTRAN now provides a graphics library which supports text manipulation under both DOS and OS/2, and line- and shape-drawing functions under DOS. For more information, see *Microsoft FORTRAN Advanced Topics*.

B.3 Changes from Version 4.0 to Version 4.1

The principal change from Version 4.0 to Version 4.1 is the addition of OS/2 support. The compiler, the linker, and the FL utility have been modified to correctly compile and link OS/2 real- or protected-mode programs.

B.3.1 OS/2 Support

You can write protected-mode programs that run under OS/2, real-mode programs that run under DOS Version 3.0 (or higher), or bound programs that run under both operating systems. For detailed information about developing programs for OS/2, see Chapter 8, "Programming for OS/2," in *Microsoft FORTRAN Advanced Topics*.

B.3.2 Enhanced FL Utility

You can now use an environment variable to specify frequently used FL options. The new version of FL also lets you specify libraries anywhere on the command line and create overlays without a separate link step. The FL command also includes switches for compiling and linking under OS/2.

B.3.2.1 Creating Overlays

The FL utility now lets you specify program overlays on the command line, by enclosing the modules to be overlaid in parentheses.

B.3.2.2 Specifying Libraries

If you have set up the compiler for both real- and protected-mode OS/2, you must specify the correct library to use when linking. You can use the /Lp option to specify the protected-mode libraries, and the /Lr or /Lc option to specify the real-mode libraries.

B.3.2.3 FL Environment Variable

The FL environment variable can be used to specify a group of default options that are automatically added to the FL command line.

B.3.2.4 Creating Bound Applications

The /Fb option may be used to create a bound program that runs under either DOS or OS/2.

B.3.2.5 Mixing .LIB and .DEF Files

You can now mix .LIB and .DEF files on the command line with other options and files. FL sorts through the file names and sends them in the correct order to the linker.

B.3.2.6 /NOI Option

The /NOI (NOIGNORECASE) option is no longer the default. You must specify this option by including /link /NOI on the command line or in the environment variable.

B.3.2.7 /Tf Option

To specify a source file with an extension other than .FOR, use the /Tf option.

B.3.3 Extended Control Over Default Libraries (Linker Options)

An extension to the /NOD option lets you specify that a particular default library not be searched during linking.

The version of the linker included with Microsoft FORTRAN Version 4.1 includes an extension to the /NOD (ignore default libraries) option. In previous versions, /NOD made the linker avoid searching any library specified in the object file. You can now use the option to tell LINK not to search a specific default library by including the name after a colon (:).

B.4 Changes from Versions 3.2 and 3.3 to Version 4.0

This section describes features of the Microsoft FORTRAN Compiler, Version 4.0, that are extensions of or changes to Version 3.3. It summarizes the changes made in Version 4.0 to support the ANSI full-language standard; it discusses compatibility between source and object files for Versions 3.2, 3.3, and 4.0; and it describes changes and additions to the compiler and linker software, the run-time library system, and the language itself.

B.4.1 Changes for ANSI Full-Language Standard

Version 4.0 of the Microsoft FORTRAN Compiler is an implementation of the ANSI X3.9-1978 FORTRAN full-language standard; Version 3.3 implemented only the subset standard. The following list summarizes the new features in Version 4.0 that were required for the ANSI full-language standard.

Language Construct	Change for Version 4.0
Concatenation operator (//)	Now supported.
Asterisk length specifiers	Can be used with character functions and character parameters.
CHARACTER*n* arguments	Argument length passed with **CHARACTER***n* arguments to subprograms or functions. The maximum value of *n* is now 32,767 instead of 127.
Format specifiers	Can be character arrays.
Unit specifiers	Unit specifiers that include the **UNIT=** keyword can appear at any position in the I/O control list.
LEN intrinsic function	Now supported.
INDEX intrinsic function	Now supported.
Assignment statement (computational)	Can include Hollerith constants.
BACKSPACE, **ENDFILE**, and **REWIND** statements	Can transfer control to a label after errors and use a variable to indicate error or end-of-file status.
CLOSE and **OPEN** statements	Can transfer control to a label after errors. The **OPEN** statement can specify how blanks are interpreted in numeric input.
DATA statement	Items that are assigned values can include substring names and implied-**DO** lists.
CALL statement	Can include Hollerith constants.
DATA statement	Can include Hollerith constants.
DIMENSION statement	Both upper and lower bounds allowed for dimension declarators.
DO statement	Loop indices can be of any **REAL** type.
ENTRY statement	Now supported.

STATUS= in **OPEN** statements	Opening existing files with the **STATUS='NEW'** option is illegal. **STATUS='UNKNOWN'** in Version 4.0 behaves the same way as does **STATUS='NEW'** in Version 3.3.
Constants in **PARAMETER** statements	Arithmetic, logical, and character constants fully supported.
PRINT statement	Now supported.
READ statement	**READ** statements without a control information list or without a unit specifier supported.

B.4.2 Source Compatibility

Version 4.0 of the Microsoft FORTRAN Compiler compiles any valid source program that you successfully compiled using an earlier version of the compiler, except where list-directed I/O and direct-access I/O are used together. However, source programs may behave differently when compiled with Version 4.0.

B.4.3 Attributes in Array Declarations

In array declarations in Version 4.0, attributes appear before the list of array bounds. In Version 3.3, attributes appear after the list of array bounds.

For example, this declaration in a Version 3.3 source file

```
DIMENSION x(10)[VALUE]
```

should appear as shown below in a Version 4.0 source file:

```
DIMENSION x[VALUE](10)
```

B.4.4 Blanks in Formatted Files

The ANSI full-language and subset standards treat blanks in formatted files differently. In the full-language standard, blanks are treated as null characters unless the **BN** and **BZ** format descriptors, or the **BLANK=** option in an **OPEN** statement, specify otherwise. In the subset standard, blanks are treated as zeros unless the **BN** and **BZ** format descriptors indicate otherwise.

Version 4.0 supports the full-language treatment of blanks: it considers blanks to be null characters unless otherwise specified.

If the files used by a program expect blanks to be treated as zeros by default, the program must include the **BLANK='ZERO'** option in the **OPEN** statements for those files.

B.4.5 MODE and STATUS Options in OPEN Statement

In Version 4.0, if the **MODE=***mode* option does not appear in an **OPEN** statement, the FORTRAN run-time system tries to open the file with **MODE** values of **'READWRITE'**, **'READ'**, and **'WRITE'**, in that order. In Version 3.3, if the **MODE=***mode* option does not appear in an **OPEN** statement, the FORTRAN run-time system tries to open the file with **MODE** values of **'READWRITE'**, **'WRITE'**, and **'READ'**, in that order.

In Version 4.0, when the **STATUS='NEW'** option appears in an **OPEN** statement, the file specified in the statement must not exist. If an existing file has the same path name as the file specified in the statement, an error results. In Version 3.3, when the **STATUS='NEW'** option appears in an **OPEN** statement, the file specified in the statement can exist at the time the statement is executed. Any file with the same path name as the file specified in the statement is overwritten. (This conflicts with a strict interpretation of the standard.)

If you want programs compiled using Version 4.0 to behave in the same way as programs compiled using Version 3.3, substitute the **STATUS='UNKNOWN'** option for the **STATUS='NEW'** option in any **OPEN** statements that specify the path names of existing files.

B.4.6 Temporary Scratch-File Names

In Version 4.0, if no file name is specified in an **OPEN** statement, the FORTRAN run-time system creates a temporary scratch file with a file name in the following format:

ZZ*processno*

In this file name, *processno* is an alphanumeric character followed by a 5-digit process number. The alphanumeric character is "0" for the first temporary file opened, followed by the letters "a", "b", "c", and so on for each subsequent file name. For example, if you opened five files with no file names in a single program, the file names assigned to the temporary files would be the following (if "12345" is the process number):

`ZZ012345` (first file opened)
`ZZa12345` (second file opened)
`ZZb12345` (third file opened)
`ZZc12345` (fourth file opened)
`ZZd12345` (fifth file opened)

In Version 3.3, if no file name is specified in an **OPEN** statement, the scratch file name has the following format:

T*unitspec***.TMP**

In this file name, *unitspec* is the unit number specified in the **OPEN** statement.

B.4.7 Binary Direct Files

In Version 4.0, binary files can be opened for direct access. In most cases, I/O operations performed on binary direct files produce the same results as the same operations performed on unformatted direct files. An exception is that the number of bytes transferred in a single binary direct read or write operation is no longer limited by the record length (although even multiples of the record length are still used in repositioning between successive **READ** and **WRITE** statements).

See Chapter 5, "Record Structure: File Formats and Access," in *Microsoft FORTRAN Advanced Topics*.

B.4.8 Precision of Floating-Point Operations

Programs that use floating-point values may give slightly different results when compiled with Version 4.0 because Version 4.0 passes more information to the 8087/80287 coprocessor than Version 3.3. This has the effect of maintaining higher precision than if the values were truncated into double- or single-precision values.

For example, in Version 4.0, arguments to transcendental functions are passed in the 8087/80287 registers. If these arguments are expressions, their values are in the 64-bit precision of the coprocessor. In Version 3.3, arguments to transcendental functions are passed in memory as either single- or double-precision values. Thus, these arguments are truncated to 23- or 52-bit precision, respectively.

See Chapter 1, "Controlling Floating-Point Operations," in *Microsoft FORTRAN Advanced Topics* for more information.

B.4.9 Exponentiation Exceptions

Versions 4.0 and 3.3 give different results for certain cases of exponentiation. These differences fall into four categories:

1. Zero raised to a zero power
2. Zero raised to a negative power
3. **COMPLEX** zero raised to a **COMPLEX** power
4. Negative **INTEGER** or **REAL** values raised to a **REAL** power

Tables B.1–B.4 summarize these differences. The following abbreviations are used in the tables:

Abbreviation	Meaning
f	Nonintegral real number (for example, 1.5)
–n	Negative integer
+r	Positive real number
–r	Negative real number
s	Nonzero real number
w	Integral real number (for example, 3.0)

Table B.1 Negative INTEGER or REAL Raised to a REAL Power

Base Type	Exponent Type	Formula	Version 4.0 Example	Returns	Version 3.3 Returns
INTEGER	**REAL**	$(-n)^w$	$(-3)^{3.0}$	–27.0	Error†
INTEGER	**REAL**	$(-n)^f$	$(-1)^{1.5}$	Error	Error†
REAL	**REAL**	$(-r)^w$	$(-3.0)^{3.0}$	–27.0	Error†
REAL	**REAL**	$(-r)^f$	$(-1.0)^{1.5}$	Error	Error†

† Version 3.3 does not allow exponentiation of a negative number to a **REAL** power. Version 4.0 allows it only if the exponent is a whole number, such as 3.0; it does not allow fractional exponents such as 1.5. These restrictions do not apply to exponentiation with a **COMPLEX** base (or exponent); for example, `COMPLEX (-1.0,0.0) 1.5` will give `(0.0,-1.0)` as the result.

Table B.2 Zero Raised to a Negative Power

Base Type	Exponent Type	Formula	Version 4.0 Returns	Version 3.3 Returns
INTEGER	**INTEGER**	0^{-n}	Error	Error
REAL	**INTEGER**	0.0^{-n}	Error	Error†
REAL	**REAL**	0.0^{-r}	Error	Error†
COMPLEX	**INTEGER**	$(0.0,0.0)^{-n}$	Error	(0.0,0.0)
COMPLEX	**REAL**	$(0.0,0.0)^{-r}$	Error	(0.0,0.0)
COMPLEX	**COMPLEX**	$(0.0,0.0)^{(-r,0)}$	Error	(0.0,0.0)

† In Version 3.3, **REAL** 0.0 raised to a negative power produces an error if exceptions are not masked with **LCWRQQ**, and infinity if exceptions are masked with **LCWRQQ**.

Table B.3 COMPLEX Zero Raised to a COMPLEX Power

Base Type	Exponent Type	Formula	Version 4.0 Returns	Version 3.3 Returns
COMPLEX	COMPLEX	$(0.0, 0.0)^{(+r,0.0)}$	(0.0,0.0)	(0.0,0.0)
COMPLEX	COMPLEX	$(0.0,0.0)^{(0.0,0.0)}$	(1.0,0.0)	(0.0,0.0)
COMPLEX	COMPLEX	$(0.0,0.0)^{(-r,0.0)}$	Error	(0.0,0.0)
COMPLEX	COMPLEX	$(0.0,0.0)^{(+r,s)}$	(0.0,0.0)	(0.0,0.0)
COMPLEX	COMPLEX	$(0.0,0.0)^{(0.0,s)}$	Error	(0.0,0.0)
COMPLEX	COMPLEX	$(0.0,0.0)^{(-r,s)}$	Error	(0.0,0.0)

Table B.4 Zero Raised to the Zero Power

Base Type	Exponent Type	Formula	Version 4.0 Returns	Version 3.3 Returns
INTEGER	INTEGER	0^0	1	1
REAL	INTEGER	0.0^0	1.0	1.0
REAL	REAL	$0.0^{0.0}$	1.0	1.0
COMPLEX	INTEGER	$(0.0,0.0)^0$	(1.0,0.0)	(0.0,0.0)
COMPLEX	REAL	$(0.0,0.0)^{0.0}$	(1.0,0.0)	(0.0,0.0)
COMPLEX	COMPLEX	$(0.0,0.0)^{(0.0,0.0)}$	(1.0,0.0)	(0.0,0.0)

B.4.10 List-Directed Output

In Version 4.0, the conventions for list-directed output have changed. The following conventions are used:

1. Integer output constants are produced with the effect of an **I11** edit descriptor. (Version 3.3 uses the **I12** edit descriptor for this.)
2. Real and double-precision constants are produced with the effect of either an **F** or an **E** edit descriptor, depending on the value of the constant *c* in the following range:

   $1<=c<10^7$

   a. If *c* is within the range, the constant is produced by using `0PF15.6` for single precision and `0PF24.15` for double precision. In Version 3.3, `0PF16.7` is used for single precision and `0PF23.14` is used for double precision.

b. If *c* is outside the range, the constant is produced using `1PE15.6E2` for single precision and `1PE24.15E3` for double precision. The value 0 is printed with this format. (In Version 3.3, `1PE14.6E3` is used for single precision and `1PE21.13E3` is used for double precision.)

The same field widths are used to force the constants in both cases to line up on a printed page.

B.4.11 DO-Loop Ranges

The code generated for **DO** loops in Version 4.0 uses the standard formula for determining the loop iteration count, which is, consequently, limited to the maximum allowable integer size. In Version 3.3, the code generated for **DO** loops allows more iterations than the maximum allowable integer value; for example, if the **$STORAGE:2** metacommand is in effect, a **DO** loop of the following form loops 65,535 times in Version 3.3 but is illegal in Version 4.0:

```
DO 200 I = -32767,32767
```

B.4.12 Object Compatibility

Sections B.3.13–B.3.15 discuss compatibility between object files compiled with Versions 4.0, 3.3, and 3.2. If possible, you should recompile programs and subprograms to take advantage of the improved code generated by Version 4.0. If you cannot do this (for example, if the source files are unavailable), you can continue to link object files generated by Version 3.3 with those generated by Version 4.0. However, you should read the information in the following paragraphs to make sure that object files compiled under the two versions link correctly.

B.4.13 Library Compatibility

If your program mixes modules compiled with Version 4.0 and modules compiled with Version 3.3, you must link them with the FORTRAN.LIB library that comes with Version 4.0 in addition to a standard FORTRAN library built by the SETUP program. The SETUP program installs the Version 4.0 FORTRAN.LIB if you request compatibility with Version 3.3 or 3.2. This library is required because the standard Version 4.0 libraries are different internally from the standard Version 3.3 and Version 3.2 libraries, and the code generated by the Version 4.0 compiler accesses these libraries differently. Thus, special interfaces are required so that the code produced by the two versions can work together.

The Version 4.0 FORTRAN.LIB library includes the interfaces required to work with Version 3.3 and Version 3.2 modules. It contains all the external interfaces supported by Version 3.3 and Version 3.2 FORTRAN.LIB. However, the interfaces in the Version 4.0 library generally use parts of the standard Version 4.0 library to perform their processing.

FORTRAN.LIB is not required if all of the object files you are linking were compiled with Version 4.0. Also, since modules compiled with Versions 3.3 and 3.2 have library search directives for FORTRAN.LIB embedded in them, you do not need to specify FORTRAN.LIB explicitly when you link. However, this library should be found in the standard place specified in the LIB environment variable.

You can use Version 3.3 and Version 3.2 modules with Version 4.0 modules that are compiled with any /FP compiler option, subject to the restrictions that apply to the Version 4.0 modules: that is, you cannot link with an alternate math library (LLIBFORA.LIB) if any of the modules contains in-line instructions.

However, you must still tell the SETUP program to include the compatibility math interfaces in the LLIBFORx.LIB library that it builds if you plan to use the library with Version 3.3 and Version 3.2 modules. The resulting program will not be affected, but the library that SETUP builds will be slightly larger. (The math interfaces are not included in FORTRAN.LIB since, unlike the standard FORTRAN libraries built by SETUP, FORTRAN.LIB is not typically associated with a particular /FP option.)

B.4.14 Mixing Version 4.0 and Version 3.3 Modules

Version 4.0 modules that are linked with Version 3.3 modules must be compiled using the large memory model. This model is the default for Version 4.0 FORTRAN programs. (See Chapter 2 of *Microsoft FORTRAN Advanced Topics* for more information about memory models.)

In most cases, the calling and argument-passing conventions are the same in Versions 3.3 and 4.0, so that routines compiled under either version can call each other freely. The only exception is the case of a Version 3.3 routine calling a Version 4.0 routine and passing a **CHARACTER*(*)** argument. (This situation is most likely to arise when a Version 3.3 program passes a subprogram as an argument to another subprogram compiled with Version 4.0.)

A routine compiled with Version 3.3 cannot call a Version 4.0 routine that has **CHARACTER*(*)** formal arguments. Version 4.0 expects the caller to specify the lengths of all such arguments in a special way. Since Version 3.3 does not support arguments of this type, Version 3.3 programs cannot pass the argument length. Any such call gives undefined results at run time. (This change was made in order to support the more powerful feature of the full ANSI FORTRAN-77 standard.) This problem does not arise in calls from Version 4.0 routines to Version 3.3 routines. Version 4.0 routines pass the length of a **CHARACTER*(*)** argument in such a way that Version 3.3 routines can safely ignore it.

NOTE *Certain additional rules apply if you are linking C modules with FORTRAN modules. Chapter 3 of Microsoft FORTRAN Advanced Topics explains these rules.*

If you compile a Version 3.3 source file that includes the **STATUS='NEW'** option and link the resulting object file with a Version 4.0 library that includes the Version 3.3 compatibility package, the **STATUS='NEW'** option is mapped to **STATUS='UNKNOWN'**. This results in behavior more similar to the Version 3.3 implementation of the **STATUS='NEW'** option.

B.4.15 Mixing Version 4.0 and Version 3.2 Modules

In general, programs can mix modules compiled with Versions 4.0 and 3.2 of Microsoft FORTRAN. However, the following considerations apply:

- All considerations that apply to mixing Version 3.3 modules with Version 4.0 modules also apply to mixing Version 3.2 modules with Version 4.0 modules.
- You must compile any Version 4.0 modules in these programs with the /Gr option to the FL command. This is because the code that Version 4.0 generates by default preserves the SI and DI registers for the duration of a subprogram, while the code that Version 3.2 generates does not. If you specify /Gr, the Version 4.0 code does not expect the SI and DI registers to be preserved.

B.5 Changes for Version 4.0

Sections B.4.1–B.4.4 discuss changes and enhancements to the Microsoft FORTRAN Compiler for Version 4.0. These changes fall under the following categories:

- Enhancements and additions to the compiler and linker
- Run-time library changes
- Language changes

B.5.1 Enhancements and Additions to the Compiler and Linker

Several features have been added to, or changed in, Version 4.0 of the Microsoft FORTRAN Compiler and the Microsoft Overlay Linker (LINK) to make them easier to use. These features should not affect your source code, but you may need to revise existing batch files or MAKE description files so that they work correctly with Version 4.0.

B.5.1.1 The FL Command

In Microsoft FORTRAN, a new command, FL, automatically executes the compiler and the linker. The options associated with this command give you considerable flexibility in controlling compilation and linking.

You can specify the /c option with the FL command to compile without linking. You can invoke the linker separately after you compile, either through FL or through the LINK command.

The FL command performs many of the same functions as any batch files that you may have created to compile and link your FORTRAN programs. It also allows you to specify on the command line all files you want to compile and link and all options for controlling the process. You can include wild-card characters in the files you specify so that you can easily compile and link more than one file. FL automatically prompts you if it cannot find a file that it needs at any point during compilation and linking. Note that you must give the entire source-file name, including the .FOR extension, to the FL command. If you do not include the .FOR extension, FL interprets the file name as an object-file name. If you wish to convert existing batch files so they compile and link correctly under Version 4.0, be sure that you substitute the appropriate FL command for any FOR1, PAS2, PAS3, and LINK commands that may have been in the original batch files.

See Chapter 7, "The FL Command," for detailed instructions on using the FL command for program compilation and linking.

B.5.1.2 Changes to the Linker

Several linker options have been added for Version 4.0. You can specify these options either by using the /link option of the FL command, or by using the LINK command if you choose to invoke the linker separately.

The following list gives the new linker options:

Option	Task
/CO	Prepares a special executable file for use with the Microsoft CodeView window-oriented debugger
/DO	Enforces the default segment-loading order for Microsoft language products (including Microsoft FORTRAN)
/E	Packs the executable file during linking
/HE	Lists all LINK command options on standard output
/I	Displays information about the effect of the linking process on standard error output

B.5.1.3 Memory Models

When you compile a program using Version 4.0 of the Microsoft FORTRAN Compiler, you can choose a memory model to be used for your program. The memory model you choose specifies how memory for the code and data in your program will be allocated. Three memory models are available: medium, large,

and huge. You choose a memory model by specifying the /AL (large), /AM (medium), or /AH (huge) option with the FL command at compile time. The default is the large memory model. (See Chapter 2 of *Microsoft FORTRAN Advanced Topics* for information on the use of memory models.)

All programs compiled with Version 3.3 of the Microsoft FORTRAN Compiler are large-model programs. The large model is the default memory model for Version 4.0.

For programs that mix modules compiled under Versions 3.3 and 4.0, Version 4.0 modules cannot be compiled using the medium memory model. If this model is used for the Version 4.0 modules, the program may produce undefined results, although it may appear to link correctly.

NOTE *Using the* ***$LARGE*** *metacommand on an entire program has the same effect as specifying the huge memory model, except that fixed-size arrays are implicitly declared with the* ***HUGE*** *attribute. The* ***$LARGE*** *metacommand is not associated with the large memory model.*

B.5.2 Run-Time Library Changes

The following changes have been made to the libraries provided with Version 4.0 of the Microsoft FORTRAN Compiler:

- The auxiliary library DECMATH.LIB, which supported an alternative floating-point format in Version 3.3, is no longer provided.
- The library structure for Version 4.0 is considerably different from the structure for Version 3.3. During installation, you can specify the memory model, the math package you wish to use, and various other options. Then the SETUP program builds a library according to your specifications. The memory-model and floating-point options you specify on the FL command line allow your program to be linked with the library you build automatically.

B.5.3 Changes to the Language

This section lists the changes made to the Microsoft FORTRAN language for Version 4.0. For each difference, a reference to the appropriate section in the documentation for Version 4.0 or Version 3.3 is given.

B.5.3.1 Underscore (_) as a Digit

In Version 4.0, the underscore is classified as a digit, which can be used as any character of a name other than the first character. An underscore cannot be used in names if the /4Ys option is used in compiling (or the **$STRICT** metacommand is in effect). In Version 3.3, the underscore (_) is classified as a special character, which cannot be used in names.

B.5.3.2 Dollar Sign ($) in Collating Sequence

In Version 4.0, the dollar sign is classified as an alphanumeric character, which can be used in names and which appears after uppercase Z in the collating sequence.

The dollar sign cannot be used as an alphanumeric character in names in the following cases:

- If the /4Ys option is used in compiling (or the **$STRICT** metacommand is in effect)
- If the name is declared using the C attribute

In Version 3.3, the dollar sign ($) is classified as a special character, which appears as the first character in the FORTRAN collating sequence.

B.5.3.3 Significant Characters in Names

In Version 4.0, only the first six characters in a name are significant, unless the /4Nt option is used in compiling or the **$NOTRUNCATE** metacommand is in effect. In this case, the first 31 characters in a name are significant.

In Version 3.3, only the first six characters in a name are significant under any circumstances.

B.5.3.4 Column Restrictions for Source Files

Version 4.0 allows source code to be in free-form format. The /4Yf option to the FL command (and the **$FREEFORM** metacommand) gives you this choice.

In Version 3.3, statements in source programs are required to obey the standard FORTRAN column restrictions.

B.5.3.5 Restrictions on Continuation Lines

In Version 4.0, limits on the number of continuation lines have been removed, unless the /4Ys option is used in compiling (or the **$STRICT** metacommand is in effect). In these cases, the compiler generates an error if a statement extends over more than 19 continuation lines or includes more than 1,320 characters.

In Version 3.3, these restrictions are always in effect.

B.5.3.6 Maximum Character-Value Length

In Version 4.0, the maximum length of character values is 32,767 characters. Character constants are effectively limited to 1,958 characters.

In Version 3.3, character values can have a maximum length of 127 characters.

B.5.3.7 Arithmetic Operations

In Version 4.0, raising a negative-value operand to an integral real power is permitted.

In Version 3.3, raising a negative-value operand to any real power produces an error.

B.5.3.8 Character Editing and Hollerith Data Types

In Version 4.0, Hollerith data types can be used with the **A** edit descriptor when an input/output list item is of type **INTEGER**, **REAL**, or **LOGICAL**.

B.5.3.9 Expressions in Substring Specifications

In Version 4.0, any type of arithmetic expression can be used to specify the first and last characters in a substring, unless the /4Ys is used in compiling (or the **$STRICT** metacommand is specified). In effect, noninteger substring expressions are truncated by an implicit use of the **INT** intrinsic function before substring operations are performed. If the /4Ys option (or **$STRICT** metacommand) appears, only integer expressions can be used to specify the first and last characters in a substring.

In Version 3.3, these restrictions are always in effect.

In Version 4.0 the compiler verifies the following relationships, where *first* is the arithmetic expression that defines the first character in the substring, *last* is the arithmetic expression that defines the last character, and *length* is the length of the character variable:

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

If either of these relationships is false and the /4Yb option is used in compiling (or the **$DEBUG** metacommand is in effect), the compiler generates an error message. If either of these relationships is false and the /4Yb option is not used (or the **$DEBUG** metacommand is not in effect), the substring is undefined.

B.5.3.10 Array Subscripts

In Version 4.0, array subscripts can be any arithmetic expression, unless the /4Ys option is used in compiling (or the **$STRICT** metacommand is specified). In effect, noninteger subscript expressions are truncated by an implicit use of the **INT** intrinsic function before subscripting operations are performed. If the /4Yb option is used in compiling (or the **$DEBUG** metacommand is specified), subscripts are checked on all arrays that are not formal arguments, and an error message is generated for invalid subscripts.

In Version 3.3, array-element references must be integer expressions.

B.5.3.11 Changes to the Input/Output System

This section describes changes to the input/output system used in Version 4.0 of Microsoft FORTRAN.

Unit Specifiers

In Version 4.0, unit specifiers can be used more flexibly. The optional **UNIT=** string can appear before the unit specifier in all I/O statements except **PRINT**, **INQUIRE** with a **FILE=** option, and the **EOF** intrinsic function. If the optional **UNIT=** string appears in the unit specifier, the specifier can appear at any position in the I/O control list. This change was made to conform with the ANSI full-language standard for FORTRAN.

In Version 3.3, the unit specifier must appear in the first position.

In Version 4.0, the following external unit specifiers can be reconnected to another file:

External Unit	Description
0	Initially represents the keyboard or the screen
5	Initially represents the keyboard
6	Initially represents the screen

If you connect any of these specifiers to a different file using an **OPEN** statement and then close that file, the specifier resumes its preconnected status.

Output Lists

In Version 4.0, arbitrary expressions used in an output list can begin with a left parenthesis.

In Version 3.3, arbitrary expressions used in an output list cannot begin with a left parenthesis because left parentheses are reserved for implied-**DO** lists.

Format Specifiers

In Version 4.0, statement labels, integer variables, character expressions, character variables, or character arrays can be used as format specifiers. If the /4Ys option is not used in compiling (and the **$STRICT** metacommand is not in effect), noncharacter arrays can also be used.

In Version 3.3, only statement labels, integer variables, character expressions, or character variables can be used as format specifiers.

Backslash (\ ) Edit Descriptor

In Version 4.0, the backslash (\ ) edit descriptor is only recognized for files connected to terminal devices such as screens or printers. Otherwise, it is ignored.

In Version 3.3, the backslash edit descriptor is recognized for all file types.

B.5.3.12 Assignment Statement (Computational)

In Version 4.0, the *expression* in a computational assignment statement can be a Hollerith constant. A Hollerith constant can be assigned to any type of *variable*. The normal rules for padding and truncation of character data types also apply to Hollerith constants.

In Version 3.3, Hollerith constants cannot be used in assignments.

B.5.3.13 CALL Statement

In Version 4.0, the *actuals* parameter can include Hollerith constants. Hollerith constants cannot be passed to character formal arguments.

In Version 3.3, Hollerith constants cannot be used in **CALL** statements.

B.5.3.14 DATA Statement

In Version 4.0, the *nlist* parameter in a **DATA** statement can include substring names and implied-**DO** lists, and the *clist* parameter can include Hollerith constants. The normal rules for padding and truncation of character data types also apply to Hollerith constants.

In Version 3.3, these constructs are not allowed.

B.5.3.15 BACKSPACE, ENDFILE, and REWIND Statements

In Version 4.0, the **BACKSPACE**, **ENDFILE**, and **REWIND** statements can include an **ERR=** option to specify the flow of control after errors, and an **IOSTAT=** option to specify a variable to be used to indicate error or end-of-file status.

In Version 3.3, the only option allowed in the **BACKSPACE**, **ENDFILE**, and **REWIND** statements is a unit specifier, which specifies the unit location of the file that the command acts on.

B.5.3.16 CLOSE and OPEN Statements

In Version 4.0, the **CLOSE** and **OPEN** statements can include the **ERR=** option to specify the flow of control if an error occurs during statement execution. In addition, the **OPEN** statement can include the **BLANK=** option to indicate how blanks are interpreted in numeric input and the **BLOCKSIZE=** option to assign a new I/O-buffer size for the file being opened.

B.5.3.17 DIMENSION Statement

In Version 4.0, no restriction is placed on the number of array dimensions unless the /4Ys option is used in compiling, or the **$STRICT** metacommand is set. In that case, arrays are restricted to seven dimensions.

Arrays in Version 3.3 are always restricted to seven dimensions.

In Version 4.0, lower array-dimension bounds can be specified explicitly and can be positive, negative, or 0. If a lower dimension bound is not specified, it is 1 by default.

The upper and lower bounds are checked according to the following rules:

- If the upper and lower dimension bounds are constants, the compiler verifies that the upper dimension bound is greater than or equal to the lower dimension bound. If it is not, the compiler generates an error message.
- If either the upper or the lower dimension bound is not a constant, the /4Yb compiler option must be used (or the **$DEBUG** metacommand must be in effect) if you want to verify that the upper bound is greater than or equal to the lower bound.

NOTE *If all of an array's dimensions are declared with no lower bounds and with upper bounds of 1, no bounds checking is performed, even if /4Yb or* ***$DEBUG*** *is used. In this case, the array is treated the same as an adjustable-size array, except that the declared size of the array is used to determined whether or not huge addressing is used.*

Dimension declarators in Version 3.3 do not include lower bounds; the lower bound is always 1.

In Version 4.0, a dimension declarator can be an arithmetic expression, unless the /4Ys option is used in compiling (or the **$STRICT** metacommand is specified). The result of the expression is truncated to an integer by an implicit use of the **INT** intrinsic function. If an arithmetic expression is used as a dimension declarator, it cannot contain function or array-element references. If a dimension declarator with variables is used to declare an adjustable-size array, the variables either must be formal arguments to a routine or must exist in a common block. Also, the array itself must be a formal argument.

B.5.3.18 DO Statement

In Version 4.0, loop indices in a **DO** statement can be integer, real, or double-precision expressions.

In Version 3.3, loop indices in a **DO** statement must be integer expressions.

B.5.3.19 INQUIRE Statement

In Version 4.0, the **INQUIRE** statement can include the **BINARY=** option to indicate whether the file (or the file connected to the unit) specified in the statement is in binary format. It can also include the **BLOCKSIZE=** option, which reports the I/O buffer size for the file.

In Version 3.3., these options do not appear.

B.5.3.20 PAUSE Statement

In Version 4.0, the **PAUSE** statement allows the user to enter a blank line to return control to the program. It also allows the user to execute one or more DOS commands before returning control to the program. If this feature is used, the subdirectory containing COMMAND.COM should be part of the user's search path. While the program is suspended, the user can enter either of the following:

- A DOS command. After the command is executed, control is automatically returned to the program.
- The word **COMMAND** (uppercase or lowercase). After entering `COMMAND`, the user can enter as many DOS commands as desired, then type `EXIT` (uppercase or lowercase) to return control to the program.

In Version 3.3, the **PAUSE** statement only allows the user to enter a blank line to return control to the program.

B.5.3.21 READ and WRITE Statements

In Version 4.0, the **READ** and **WRITE** statements can include the **FMT=** *formatspec* option, which can appear at any position in the I/O control list. However, the **READ** statement must include a unit specifier if the **FMT=***formatspec* option is used.

In Version 3.3, a format specifier must be the second argument in a formatted **READ** or **WRITE** statement.

In Version 4.0, the unit specifier can be omitted in a **READ** statement of the following form:

READ *formatspec, iolist*

In this form of the **READ** statement, the unit is assumed to be the keyboard (*) unit.

In Version 3.3, a unit specifier must be the first argument to a **READ** statement.

B.5.3.22 STOP Statement

In Version 4.0, if the *message* parameter in a **STOP** statement is an integer, the program displays this value on the screen and returns the least-significant byte of this value to the operating system. (This is a value between 0 and 255, inclusive.) If the *message* parameter is not an integer, the program displays this value on the screen and returns 0 to the operating system.

In Version 3.3, if the *message* parameter in a **STOP** statement is an integer, the program displays the specified integer.

B.5.3.23 Type Statements

Type statements in Version 4.0 can be used to initialize the values of variables. However, variables that appear in **COMMON** and **EQUIVALENCE** statements cannot be initialized in this way.

Also, length specifiers in type statements in Version 4.0 can appear either before or after dimension declarators.

B.5.3.24 Conditional Compilation

In Version 4.0, the /4cc option of the FL command (or the **$DEBUG:***string* metacommand) can be used to specify conditional compilation. If one or more letters follows the /4cc option (or **$DEBUG** metacommand), lines in the source file that have one of those letters in column 1 are compiled into the program. Lines beginning with other characters are treated as comments.

B.5.4 New Language Features

Sections B.4.4.1–B.4.4.9 discuss new features for Version 4.0 of Microsoft FORTRAN and the changes you may have to make to source programs to take advantage of these features.

B.5.4.1 INTEGER*1 and LOGICAL*1 Data Types

Version 4.0 supports two new data types: **INTEGER*1** and **LOGICAL*1**.

An **INTEGER*1** value occupies 1 byte and can be any number in the range –127 to 127, inclusive. In an arithmetic expression, **INTEGER*1** is the lowest-ranked operand. If an **INTEGER*1** value is converted to an **INTEGER*2** value, the **INTEGER*1** value is used as the least-significant part of the **INTEGER*2** value, and the most-significant part is filled with copies of the sign bit (that is, it is sign extended). A new intrinsic function, **INT1**, is provided to convert values to type **INTEGER*1**.

A **LOGICAL*1** value occupies 1 byte of storage. The value of this byte is either 0 (**.FALSE.**) or 1 (**.TRUE.**).

B.5.4.2 C Strings

The following new string escape sequences from the C language have been added for Version 4.0:

Sequence	Character
\""	Double quote
\ x*hh*	Hexadecimal bit pattern (where *h* is between 0 and F, inclusive)
\ a	Bell

See Chapter 2 in *Microsoft FORTRAN Advanced Topics,* for more information about C strings.

B.5.4.3 Concatenation Operator

Version 4.0 supports the use of the concatenation operator (//) in character expressions.

B.5.4.4 New Intrinsic Functions

New intrinsic functions that perform data-type conversion and bit manipulation have been added for Version 4.0.

Data-Type Conversion

The following list summarizes the new intrinsic functions that are used for data-type conversion:

Function	Operation
INT1	Converts arguments to type **INTEGER*1**
HFIX	Converts arguments to type **INTEGER*2**
JFIX	Converts arguments to type **INTEGER*4**

See Chapter 5, "Intrinsic Functions and Additional Procedures," for more information about these functions.

Bit Manipulation

In Version 4.0, several new intrinsic functions can be used to perform bit-wise operations on variables. The list below summarizes these new intrinsic functions:

Function	Operation
BTEST	Bit test
IAND	Logical product
IBCHNG	Bit change
IBCLR	Bit clear
IBSET	Bit set
IEOR	Exclusive or
IOR	Inclusive or
ISHA	Arithmetic shift
ISHC	Rotate
ISHFT	Logical shift
ISHL	Logical shift
NOT	Logical complement

All of these functions except **NOT** and **BTEST** accept two arguments of type **INTEGER**, **INTEGER*1**, **INTEGER*2**, or **INTEGER*4** and return a result of the same type. If two arguments with different **INTEGER** types are given, the larger of the two types is returned (provided that it is also a legal type).

NOT accepts one argument of one of these types and returns a result of the same type. **BTEST** accepts two arguments of one of these types and returns a **LOGICAL** result. All of these functions can be passed as actual arguments.

B.5.4.5 New Time and Date Functions

New subroutines and functions that get and set the date and time have been added for Version 4.0. The following list summarizes these functions:

Function	Operation
GETDAT	Gets the system date
GETTIM	Gets the system time
SETDAT	Sets the system date
SETTIM	Sets the system time

See Chapter 5, "Intrinsic Functions and Additional Procedures," for more information about these functions.

B.5.4.6 Z Edit Descriptor

The new **Z** repeatable edit descriptor allows you to specify hexadecimal editing in input/output lists. This edit descriptor has the form **Z***w*, which specifies a field that is w characters wide. Hexadecimal digits A–F are output in uppercase. See Section 3.7.2 for rules for the use of this edit descriptor.

B.5.4.7 ENTRY Statement

ENTRY specifies an entry point for a subroutine or external function.

B.5.4.8 PRINT Statement

PRINT specifies output to the screen (unit *).

B.5.4.9 $[NO]DECLARE, $[NO]FREEFORM, and $[NO]TRUNCATE Metacommands

Six new metacommands have been added to Version 4.0 of Microsoft FORTRAN **$DECLARE**, **$NODECLARE**, **$FREEFORM**, **$NOFREEFORM**, **$TRUNCATE**, and **$NOTRUNCATE**.

The **$DECLARE** metacommand causes the compiler to display warning messages for variables that are not declared in type statements. The **$NODECLARE** metacommand suppresses these warnings. The **$NODECLARE** metacommand is the default. Note that the /4Yd compiler option has the same effect as the **$DECLARE** metacommand, and the /4Nd compiler option has the same effect as the **$NODECLARE** metacommand.

The **$FREEFORM** metacommand tells the compiler that the source program ignores the standard FORTRAN column restrictions (labels in columns 1–5, continuation characters in column 6, statements in columns 7–72, and any columns beyond 72 ignored). The **$NOFREEFORM** metacommand tells the compiler that the source program observes these column restrictions. **$NOFREEFORM** is the default. Note that the /4Yf compiler option has the same effect as the **$FREEFORM** metacommand, and the /4Nf compiler option has the same effect as the **$NOFREEFORM** metacommand.

The **$TRUNCATE** metacommand tells the compiler to generate warning messages for any names longer than six characters. This option makes it easier to port your programs to other systems. The **$NOTRUNCATE** metacommand tells the compiler to treat the first 31 characters in a name as significant. **$TRUNCATE** is the default. Note that the /4Yt compiler option has the same effect as the **$TRUNCATE** metacommand, and the /4Nt compiler option has the same effect as the **$NOTRUNCATE** metacommand.

See Chapter 6, "Metacommands," for more information on these metacommands.

Appendix C

Compiler and Linker Limits

This appendix discusses the limits imposed by the Microsoft FORTRAN Compiler and the Microsoft Segmented-Executable Linker (for example, the maximum length of an identifier) and suggests programming strategies for avoiding these limits.

C.1 Compiler Limits

To operate the Microsoft FORTRAN Compiler you must have sufficient disk space for the compiler to create temporary files used in processing. The required space is approximately two times the size of the source file.

Table C.1 summarizes the limits imposed by the Microsoft FORTRAN Compiler. If your program exceeds one of these limits, an error message will inform you of the problem.

Table C.1 Limits Imposed by the Microsoft FORTRAN Compiler

Program Item	Maximum Limit
Actual arguments	Number per subprogram: approximately 64
Character constants	Length: approximately 1,900 bytes
Names	Length: 31 bytes (default) or 6 bytes (if /4Ys or /4Yt is used in compiling or if **$STRICT** or **$TRUNCATE** metacommand is in effect); in either case, additional characters are discarded
	Internal, length: 40 bytes per module: 20,000 names
Simple variables	Number of simple variables per subprogram: approximately 5,000 (depending on lengths of variable names)
Statements	Levels of nesting: approximately 40 levels
ENTRY statements	Number per subroutine: 32,000
FORMAT statements	Number per module: 20,000
	Format length: approximately 1,900 characters
	Memory limitations: in medium-model programs, no more than 64K internal formats in the default data segment
	Number of errors per statement: 10 errors

Table C.1 *(continued)*

Program Item	Maximum Limit
GOTO statements (assigned)	Number per subroutine: 64
INTEGER items	Size: 128 bytes for a string of digits
Include files	Levels of nesting: 10 levels

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

During compilation, large programs are most often limited in the number of identifiers allowed in any one source file. They are also occasionally limited by the complexity of the program or one of its statements.

C.1.1 Limits on Number of Names

The Microsoft FORTRAN Compiler limits the number of names you can use in a source program. The compiler creates symbol-table entries for the names declared in source programs. Symbol-table entries are created for the following objects:

- The program
- Subroutines and functions declared or referenced in the program unit
- Common blocks and variables
- Statement functions
- Formal parameters
- Local variables

Common variables, statement functions, formal parameters, and local variables are required only while the subroutine or function containing them is being compiled. These names are discarded at the end of the subroutine, and the space they used is made available for other names. Hence, you can create much bigger programs by splitting up your code into more subroutines and functions so that the space for local names can be shared. You can also place the subroutines and functions into their own files and compile them separately, since this usually reduces the number of names in groups being used per module.

C.1.2 Limits on Complicated Expressions

The compiler may run out of memory when it encounters any of the following:

- A deeply nested statement or expression
- A large number of error messages
- A large block of specification statements (**EQUIVALENCE** statements in particular)

Usually, if pass 1 runs successfully on a program without running out of memory, pass 2 will also run successfully, except for complicated basic blocks. A basic block is defined as follows:

- A sequence of statements with no labels or other breaks
- A sequence of statements containing long expressions or parameter lists (especially including I/O statements or character expressions)

Pass 2 makes a smaller number of symbol-table entries than pass 1 (for example, for the program, subroutines, and functions declared or referenced in the program unit, for common blocks, and for many of the transcendental functions called in a program). If pass 2 runs out of memory, it displays a line-number reference and one of the following messages:

```
out of heap space
```

```
expression too complex, please simplify
```

If a particularly long expression or parameter list appears near this line, break up the expression or parameter list by assigning parts of the expression to local variables or by using multiple **WRITE** statements. If this does not work, add labels to statements to break the basic block.

C.1.3 Limits on Character Expressions

Use the following programming strategies to avoid compiler limitations when initializing or assigning values to large character variables or array elements:

- Use smaller pieces
- Use substrings
- Use **EQUIVALENCE** statements to assign values to a character array

To avoid compiler limitations on character expressions, assign pieces of the character value to smaller variables or substrings. Just having nonconstants in the expression causes more of the expression to be evaluated at run time instead of at compile time, thus avoiding the 1,900-character compile-time limit on constants.

C.2 Linker Limits

Table C.2 summarizes the limits imposed by the linker. If you encounter one of these limits, you must adjust your program so that the linker can accommodate it.

Table C.2 Limits Imposed by the Microsoft Segmented-Executable Linker

Item	Limit
Symbol table	Only limited by available memory
DOS load-time relocations	Default is 32K. If /EXEPACK is used, the maximum is 512K.
External symbols per module	1,023
Groups	Maximum number is 21, but the linker always defines DGROUP so the effective maximum is 20.
Overlays	63
Logical segments	128 by default; however, this maximum can be set by using the **/SEGMENTS** option of the **LINK** command.
Libraries	32
Group definitions per module	21
Physical segments per module	255
Stack	64K

C.3 Run-Time Limits

When running under MS-DOS or OS/2, a FORTRAN program cannot open more than 20 files at one time. If an OS/2 program uses either the multithread library or a dynamically-linked FORTRAN run-time library, this limit is extended to 40 files. In practice, the actual limit might be slightly less, depending on how the operating system uses available memory.

Exceeding this limit will halt program execution and produce a run-time error message. The message varies, depending on whether you have exceeded the run-time limit set by FORTRAN, or the operating system limit on the number of open files.

You can increase the maximum number of open files with the following procedures. In general, you should increase both the FORTRAN run-time limit and the operating system limit; if you have exceeded the FORTRAN run-time limit, increasing just the operating system limit will have no effect.

C.3.1 Increasing the Maximum Number of Open Files

FORTRAN 5.0 allows you to increase the maximum number of open files. To do this, you must be running under DOS 3.3 (or a later version), or under OS/2 (any version).

The following instructions refer to two .ASM files. These files are included on your Microsoft FORTRAN distribution disks in directories called `STARTUP`, `STARTUP\DOS` and `STARTUP\OS2`, whose location is specified in the PACKING.LST file on the SETUP disk. The appropriate .ASM files for each operating system are in the corresponding subdirectory. Both multithread and dynamically-linked FORTRAN run-time libraries require special considerations (See Section C.3.3, below).

C.3.1.1 Increasing File Handles

Edit the startup source file CRT0DAT.ASM to increase the number of file handles. Change the number in the line

```
_NFILE_ = 20
```

to the maximum number of file handles desired. The limit is 256.

C.3.1.2 Increasing Units

The next step is to increase the size of the table which FORTRAN uses to manage units. Edit the source file UNIT.ASM so that the number in the line

```
_NFILE_ = 20
```

equals the same value you chose in CRT0DAT.ASM. Note that the number of handles must always be greater than or equal to the number of units. Therefore, if you increase the number of units in UNIT.ASM, there must be at least as many file handles specified in CRT0DAT.ASM.

C.3.1.3 Increasing the System Limit

To have more than 20 files open at one time, you must increase the file limit that the operating system imposes on your process. To do this, you must increase both the system-wide limit and the per-process limit.

C.3.1.4 Increasing the System-Wide Limit

You can increase the system limit on the number of open files by changing the **FILES** specification in the CONFIG.SYS file. If you wanted 100 files, you would place the following statement in your CONFIG.SYS file (or change the existing **FILES** statement):

```
FILES=100
```

C.3.1.5 Increasing the Per-Process Limit

You must also increase the number of files the operating system makes available for your particular process. This is done by enabling the appropriate commented-out code in CRT0DAT.ASM.

For example, the MS-DOS version of CRT0DAT.ASM contains the following commented-out code:

```
;       mov       ah, 67H
;       mov bx,   _NFILE_
;       callos
```

In the OS/2 version of CRT0DAT.ASM, this code appears as a call to DOSSETMAXFH. Under OS/2, you must also enable the 'extern DOSSETMAXFH:far' declaration that appears near the beginning of the file.

In either case, remove the semicolons to enable this code.

C.3.2 Using the Modified Files

After modifying CRT0DAT.ASM and UNIT.ASM, assemble the files, using the batch files and make files in the `STARTUP` directory on the distribution disks. To use the new object code, either explicitly link your program with the new CRT0DAT.OBJ and UNIT.OBJ files, or replace the CRT0DAT.OBJ and UNIT.OBJ objects in the appropriate memory-model version of the FORTRAN run-time library.

C.3.3 Multithread and Dynamic Link Applications

To increase the default number of files that may be opened when linking with the multithread or dynamic link libraries from 40, you need only edit the UNIT.ASM file. Set the _NFILE_ constant defined between "else" and "endif" to the desired number of files, as described above. When assembling UNIT.ASM, be sure to define MTHREAD on the command line in the startup makefile to assure the correct conditional assembly:

```
masm -DMTHREAD unit. asm
```

You do not need to edit CRT0DAT.ASM. Instead, you must make an explicit call to DOSSETMAXFH in your application. The form of the call is shown in the OS/2 version of CRT0DAT.ASM. You must also increase the system-wide limit, as explained above.

Appendix D

Error Messages

This appendix lists error messages you may encounter as you develop a program, and describes actions you can take to correct the errors. The list below indicates where to find error messages for various components of Microsoft FORTRAN:

Component	Section
Command line used to invoke the Microsoft FORTRAN Compiler	Section D.1, "Command-Line Error Messages"
Microsoft FORTRAN Compiler	Section D.2, "Compiler Error Messages"
Microsoft FORTRAN run-time libraries and other run-time situations	Section D.3, "Run-Time Error Messages"

D.1 Command-Line Error Messages

Messages that indicate errors on the command line used to invoke the compiler have one of the following formats:

```
command line fatal error D1xxx: messagetext
command line error D2xxx: messagetext
command line warning D4xxx: messagetext
```

If possible, the compiler continues operation, printing error and warning messages. In some cases, command-line errors are fatal and the compiler terminates processing. The following messages indicate errors on the command line:

Number Command-Line Error Message

D2000 **UNKNOWN COMMAND-LINE ERROR**
Contact Microsoft Product Support Services

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

D2001 **too many symbols predefined with /D**

The number of predefined symbols exceeded the limit of 30 on the CL command line, or the limit of 20 on the FL command line.

Check the CL or FL environment variable for option specifications.

Number	Command-Line Error Message

D2002 **memory-model conflict**

More than one memory-model option was specified.

For example, the following command line causes this error:

```
cl /AS /AM program.c
```

Check the CL or FL environment variable for option specifications.

D2003 **missing source filename**

The command line did not specify a source file.

D2008 **limit of *option* exceeded at *string***

The given option was specified too many times. The given string is the argument to the option that caused the error.

If the environment variable CL or FL is set, the compiler reads options specified in the environment variable before options specified on the command line.

D2011 **only one floating-point option allowed**

More than one floating-point (/FP) option was specified on the command line.

D2012 **too many linker arguments**

More than 128 options and object files were passed to the linker.

D2013 **incomplete model specification**

Not enough characters were given for the /Astring option.

Two types of options begin with /A:

- The /Astring customized memory-model option requires three letters in *string*. The letters specify the code-pointer size, data-pointer size, and data-segment setup attributes.
- The /Ax standard memory-model option requires one uppercase letter for *x*. CL interprets a lowercase letter as part of a customized memory-model specification.

Number	Command-Line Error Message
D2013	*(continued)*

For example, the following command line causes this error:

```
cl /As file1.c
```

D2016 *option1* **and** *option2* **are incompatible**

The given command-line options cannot be specified together.

For example, the following command line causes this error:

```
cl /Gw /NDxx program.c
```

In this example, the /Gw and /NDxx options are incompatible because each has a different special-entry sequence.

D2018 **cannot create linker response file**

The compiler could not create a response file for passing arguments to to the linker.

This error can occur when an existing read-only file has the same name as the filename the compiler gives to the response file.

D2019 **cannot overwrite source or object file** *filename*

A source or object filename was specified as an output file with the /Fo option. The compiler cannot overwrite input files.

D2020 *option* **option requires extended keywords to be enabled (/Ze)**

The /Gc or /Gr option was specified on the same command line as the /Za option.

The /Gc and /Gr options require the extended keyword **_cdecl** to be enabled. To enable **_cdecl** and make library functions accessible, specify the /Ze option.

D2021 **invalid numeric argument** *number*

A number greater than 65,534 was specified as a numeric argument.

D2022 **cannot open** *messagefile*

The given file was not in the current directory or a directory specified in the PATH environment variable. The file contains a brief summary of compiler command-line syntax and options.

Number	Command-Line Error Message

D2027 **cannot execute** *component*

The compiler could not run the given compiler component or linker.

One of the following may have occurred:

- There was not enough memory to load the component. If this occurred when the compiler was invoked by NMAKE, use the DOS utility NMK or run the compiler outside of the makefile.
- The component was for another operating system.
- The component was corrupted.
- An option was specified incorrectly. For example, the following CL command causes this error:

```
cl /B1 file1.c
```

D2028 **too many open files, cannot redirect** *filename*

Redirection of one of the standard stream files was not possible because too many files were already open and a duplicate handle could not be created.

D2030 **INTERNAL COMPILER ERROR in** *component*
Contact Microsoft Product Support Services

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

D2031 **too many command-line arguments**

More than 128 arguments were specified to the compiler.

One cause of this error is to specify the argument `*.*` while in a large directory.

D4000 **UNKNOWN COMMAND-LINE WARNING**
Contact Microsoft Product Support Services

Note the circumstances of the warning and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

D4001 **listing overrides assembly output**

An assembly listing was not generated because another listing option (/Fc or /Fl) was specified. The other option took effect.

Number	Command-Line Error Message
D4001	*(continued)* Check the CL or FL environment variable for option specifications. To create both listings, compile separately with each option. For example, the following command line causes this warning: `cl /Fc /Fa program.c`
D4002	**ignoring unknown option** *option* The compiler did not recognize the given command-line option.
D4003	**processor-option conflict** More than one /Gn option was specified with conflicting values for *n*. The compiler used the last one specified on the command line. Check the CL or FL environment variable for option specifications. The following example command line causes this warning: `cl /G2 /G0 program.c` In this example, the compiler assumed /G0.
D4005	**cannot find** *component*; **Please enter new filename (full path) or CTRL+C to quit:** The compiler was unable to find the given component in the current directory or in a directory in the PATH environment variable.
D4007	*option1* **requires** *option2*; **option ignored** An option was specified without a required related option. The compiler ignored *option1*. One way this warning occurs is when the /C option is specified when the /c option is meant. CL options are case sensitive.
D4008	**nonstandard model; assuming large model** A character other than M, L, or H was specified with FL's /A option. FL assumed /AL.
D4009	**threshold only for far or huge data, ignored** The /Gt option cannot be used in memory models that have a single data segment. Only compact, large, and huge models support /Gt. Check the CL or FL environment variable for option specifications.

Number	Command-Line Error Message

D4011 **preprocessing overrides source listing**

A source listing was not generated because a preprocessor listing option was specified.

Check the CL or FL environment variable for option specifications.

D4012 **function declarations override source listing**

A source listing was not generated because function prototype declarations were asked for.

Check the CL or FL environment variable for option specifications.

D4013 **combined listing overrides object listing**

When /Fc is specified along with /Fl, the combined listing specified by /Fc is created.

Check the CL or FL environment variable for option specifications. To create both listings, compile separately with each option.

D4014 **invalid value** *number1* **for** *option*; **assuming** *number2*

The given option was specified with an invalid numeric argument. The compiler assumed the value *number2*.

For example, the following command line causes this warning:

```
cl /Zp3 program.c
```

In this example, 3 is an invalid value. Valid arguments for the /Zp option are 1, 2, and 4.

D4018 **.DEF files supported for segmented executable files only**

A module-definition file was specified on the command line, but an /Lr or /Lc option was also specified. The /Lr and /Lc options are used to create DOS executable files. Module-definition files are used to create OS/2 or Windows applications or DLLs.

D4019 **string too long; truncated to** *number* **characters**

A string longer than 40 characters was specified with the /ND, /NM, /NT, /Ss, or /St option. The compiler truncated the string.

D4020 *option* **: missing argument; option ignored**

A command-line option required an argument, but nothing was specified. CL ignored the option.

D.2 Compiler Error Messages

The error messages produced by the Microsoft FORTRAN Compiler fall into the following four categories:

1. Fatal error messages
2. Compilation error messages
3. Recoverable error messages
4. Warning messages

The messages for each category are listed in Sections D.2.1–D.2.4 in numerical order, with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number. All messages give the file name and line number where the error occurs. The following paragraphs discuss error-message format.

Fatal Error Messages

Fatal error messages indicate a severe problem, one that prevents the compiler from processing your program any further. These messages have the following format:

filename`(`*line*`) : fatal error F1`*xxx*`:` *messagetext*

After the compiler displays a fatal error message, it terminates without producing an object file or checking for further errors.

Compilation Error Messages

Compilation error messages identify actual program errors. These messages appear in the following format:

filename `(line) : error F2xxx: messagetext`

The compiler does not produce an object file for a source file that has compilation errors in the program. When the compiler encounters such errors, it attempts to recover from the error. If possible, it continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Recoverable Error Messages

Recoverable error messages are informational only; they do not prevent compiling and linking. These messages appear in the following format:

filename `(line) : error F3xxx: messagetext`

Recoverable error messages are similar to warning error messages (see below), except that you cannot suppress them using the /W option. (See Section 7.20.2 for a description of this option.)

Warning Messages

Warning messages are informational only; they do not prevent compilation and linking. These messages appear in the following format:

filename`(line) : warning F4xxx : messagetext`

You can use the /W option to control the level of warnings that the compiler generates.

D.2.1 Compiler Fatal Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it stops processing after printing the error message.

Number	Compiler Fatal Error Message
F1000	**UNKNOWN FATAL ERROR** **Contact Microsoft Technical Support** An unknown fatal error has occurred. Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F1001	**Internal Compiler Error** **compiler file '***filename***,' line '***number***' Contact Microsoft Technical Support** An internal compiler error has occurred. Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F1002	**out of heap space** The compiler ran out of dynamic memory space. This usually means that your program has many complex expressions. Try breaking expressions into smaller subexpressions.
F1003	**error count exceeds** *number*; **stopping compilation** The limit for compilation errors was exceeded.
F1005	**string too big for buffer** A string in a compiler intermediate file overflowed a buffer.

Number	Compiler Fatal Error Message
F1005	*(continued)*
	This internal compiler error could result from initializing a very long character string with a **DATA** statement. Try decreasing the length of the character string.
F1006	**write error on compiler intermediate file**
	A write error occurred on a compiler intermediate file.
	This could be caused by a faulty disk.
F1027	**DGROUP data allocation exceeds 64K**
	Allocation of variables to the default data segment exceeded 64K.
	For large- and huge-model programs, compile with the /Gt option to move items into separate segments.
F1031	**limit exceeded for nesting function calls**
	Function calls were nested to more than 30 levels.
F1032	**cannot open object listing file** *name*
	The compiler could not open the given object-listing file for writing.
	The file or disk may be write-protected, or the disk is full.
F1033	**cannot open assembly language output file** *name*
	The compiler could not open the given assembly-listing file for writing.
	The file or disk may be write-protected, or the disk is full.
F1035	**expression too complex, please simplify**
	The compiler could not generate code for a complicated expression.
	Try breaking the expression into simpler subexpressions and recompiling. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F1036	**cannot open source listing file** *name*
	The compiler could not open the given source-listing file for writing.
	The file or disk may be write-protected, or the disk is full.
F1037	**cannot open object file** *name*
	The compiler could not open the given object file for writing.
	The file or disk may be write-protected, or the disk is full.

Number	Compiler Fatal Error Message

F1039 **unrecoverable heap overflow in Pass 3**

The compiler ran out of dynamic memory space.

A subroutine may have too many symbols; simplify the subroutine and make it smaller.

F1041 **cannot open compiler intermediate file - no more files**

The compiler was unable to open an intermediate file because too many files were already open because the FILES=*n* setting in the CONFIG.SYS file. A setting of 30 is recommended.

F1043 **cannot open compiler intermediate file**

The compiler was unable to open an intermediate file. This could occur if the environment variable TMP was set to a nonexistent directory.

Try setting the environment variable TMP to an existing directory, or not setting TMP at all.

F1044 **out of disk space for compiler intermediate file**

The compiler ran out of disk space while writing to an intermediate file.

Try making more disk space available and recompiling.

F1045 **floating-point overflow**

A compile-time evaluation of a floating or complex expression resulted in overflow, as shown in the following example:

```
real a,b,c
   a=10e30
   b=10e30
   c=a*b
```

F1050 *name* **: code segment too large**

The amount of object code in the named segment was larger than 64K.

F1051 **program too complex**

Your program caused the compiler to overflow one of its internal tables. For example, this error can occur if your program has too many labels.

The /4Yb compiler option and the **$DEBUG** metacommand cause a large number of labels to be generated. If you encounter this message, try recompiling with the /4Nb option or changing the **$DEBUG** metacommand to **$NODEBUG** in your source file and recompiling; or, if your file contains more than one procedure, try compiling the procedures in separate files.

Number	Compiler Fatal Error Message
F1900	**maximum memory-allocation size exceeded**

The program tried to allocate more than approximately 1900 bytes at one time. This is the upper limit for the size of character constants. (See Appendix C, "Compiler and Linker Limits," for more information.)

F1901 **program too large for memory**

The combination of heap space and stack space overflowed the memory configurations of the machine.

F1902 **statement stack underflow**

This is an internal error. The compiler could not interpret the nesting of statements.

Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.

F1903 **statement-nesting limit exceeded**

Structured statements were nested too deeply.

The maximum legal depth is about 40 statements and varies slightly depending on the type of statement.

F1904 **illegal command-line option**

This error should never occur.

If it does, please report it to Microsoft Corporation using the Software Problem Report form at the back of this manual.

F1907 **too many symbols**

The program overflowed the internal symbol counter.

There is no set upper limit on the number of symbols allowed in a source file. However, in any case, no more than 20,000 names are allowed in one module.

F1908 **ASSIGN : too many format labels**

The program overflowed the assigned format-label table.

This error probably occurred because an **INTEGER*1** variable, which has a limit of 127 labels, was specified. To avoid this error, use an **INTEGER*2** variable instead. (See Appendix C, "Compiler and Linker Limits," for more information.)

F1909 *filename* **: include file nested too deeply**

More than 10 include files were active at the same time. (See Appendix C, "Compiler and Linker Limits," for more information.)

Number	Compiler Fatal Error Message
F1910	*name* **: unrecognized option**

This is an internal error; the compiler driver FL caught an illegal option.

F1912 *filename* **: cannot open file**

The specified file could not be opened.

F1913 *name* **: name too long**

The specified internal file name was more than 14 characters long. The compiler creates internal files in the directory specified by the TMP environment variable. If the combined length of the TMP environment variable and the unique internal name exceeds the name-buffer length, this message appears (see Appendix C, "Compiler and Linker Limits," for more information).

Create a smaller TMP environment variable. If no TMP variable is specified, this error should never occur.

F1914 **cannot open internal files**

Internal files could not be created.

F1917 **unknown primitive type**

An internal error has occurred.

Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.

F1918 **missing symbol reference**

An internal error has occurred.

Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.

F1919 **unknown constant type**

An internal error has occurred.

Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.

F1920 **illegal -A option**

An invalid memory-model option was given for the FL command line.

F1921 **too many ENTRY statements**

More than 32,000 **ENTRY** statements were used in this subprogram. (This error is not likely to occur.)

Number	Compiler Fatal Error Message
F1922	**integer string too long**

An integer-constant string of digits overflowed an internal buffer. (This error should never occur in normal use.)

Shorten the integer strings to legal value.

F1923 **CHARACTER constant too long**

A constant of type **CHARACTER** can have a maximum of approximately 1,900 characters. (See Appendix C, "Compiler and Linker Limits," for more information.)

F1924 **FORMAT string too long**

A **FORMAT** statement can have a maximum of approximately 1,900 characters. (See Appendix C, "Compiler and Linker Limits," for more information.)

F1925 **out of disk space for compiler internal file**

The disk became full while the compiler was writing to an internal file.

F1926 **write error on compiler internal file**

An error occurred while the compiler was writing to an internal file.

Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.

D.2.2 Compilation Error Messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues compiling the program (if possible) and outputs additional error messages. However, no object file is produced.

Number	Compiler Compilation Error Message
F2000	**UNKNOWN ERROR** **Contact Microsoft Technical Support**

An unknown compilation error has occurred.

Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.

F2001 **INTEGER value overflow**

An **INTEGER** constant was too large to be of the specified type.

Number	Compiler Compilation Error Message

F2002 **Hollerith not allowed**

Hollerith constants are not allowed when the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect).

F2003 **illegal base value**

The specified base value was not between 2 and 36, inclusive.

F2004 **INTEGER constant must follow #**

No alphanumerics within the legal range for the base immediately (allowing for white space) followed the number sign (#).

F2005 **illegal REAL constant**

A floating-point constant was in an illegal form.

F2006 **missing] following attribute string**

The closing bracket for an attribute list was missing.

F2007 **opening quote missing**

The leading quote in a string value of an **ALIAS** attribute was missing.

F2008 **unrecognized attribute**

An item used as an attribute was not a legal FORTRAN attribute.

F2009 *character* **: illegal separator**

An attribute list did not end with a closing right bracket (]) and was not continued with a comma (,), or an illegal character was used in the list for the **$NOTLARGE** metacommand.

F2010 *name* **: name too long; truncated**

The specified name was more than 31 characters long. (The limit on the length of names may be less in some environments; see Appendix C, "Compiler and Linker Limits," for more information.)

F2011 **octal value too big for byte**

An octal value was not within the range 8#0 to 8#377.

F2012 *name* **: already specified in $[NOT]LARGE**

The specified item appeared in the lists for both the **$LARGE** and the **$NOTLARGE** metacommands. (The message shows the metacommand that appears first in the source program.)

Number	Compiler Compilation Error Message
F2013	**too many continuation lines**
	Either the /4Ys compiler option was used in compiling or the **$STRICT** metacommand was in effect, and more than 19 continuation lines were used.
F2015	**$DEBUG:'<debug list>' : string expected**
	A quoted string was expected after a **$DEBUG** metacommand.
F2016	**$IF: no matching $ENDIF**
	No matching **ENDIF** was found when an **$IF** was used.
F2017	**$INCLUDE:'<filename>' : string expected**
	A quoted string specifying a file name was expected after an **$INCLUDE** metacommand.
F2018	**$LINESIZE ($PACK or $PAGESIZE) : integer constant out of range**
	An integer constant less than 80 or greater than 132 was specified in a **$LINESIZE** metacommand, or a lower bound of less than 15 was specified in the **$PAGESIZE** metacommand. An integer constant other than 1, 2, or 4, was specified for the **$PACK** metacommand.
F2019	**$LINESIZE (or $PAGESIZE) : integer constant expected**
	An integer constant was expected after a **$LINESIZE** or **$PAGESIZE** metacommand.
F2020	**$[NOT]LARGE already set**
	The **$LARGE** or **$NOTLARGE** metacommand appeared more than once in a procedure. (The metacommand appearing more than once is indicated in the message.)
F2021	**$[NOT]LARGE illegal in executable statements**
	The **$LARGE** or **$NOTLARGE** metacommand appeared between subprograms or within the specification section of a subprogram. (The metacommand that appeared between subprograms or within the specification section is indicated in the message.)
F2022	**$MESSAGE:'<message>' : string expected**
	A quoted string containing a message was expected after a **$MESSAGE** metacommand.

Number	Compiler Compilation Error Message
F2023	**divide by 0**
	The second operand in a division operation (/) evaluated to 0, giving undefined results.
F2024	**mod by 0**
	The second operand in a **MOD** function evaluated to 0, giving undefined results.
F2025	**no matching $IF**
	A **$ELSE**, **$ELSEIF**, or **$ENDIF** was seen without a previous **$IF**.
F2026	**metacommands are nonstandard**
	The metacommands used are nonstandard.
F2027	**$STORAGE:<number> : 2 or 4 expected**
	A number other than 2 or 4 followed the **$STORAGE:** metacommand.
F2028	**$SUBTITLE:'<subtitle>' : string expected**
	A quoted string was expected following a **$SUBTITLE:** metacommand.
F2029	**$TITLE:'<title>' : string expected**
	A quoted string was expected following a **$TITLE:** metacommand.
F2030	**unrecognized metacommand**
	An unrecognized string followed the dollar sign ($) in the source file.
F2031	**closing quote missing**
	A quoted string did not end with a single quote (').
F2032	**zero-length CHARACTER constant**
	An illegal **CHARACTER** constant of length 0 was used in the program.
F2033	**Hollerith constant exceeds 1313 characters**
	A Hollerith constant exceeded the maximum legal length.
F2034	**zero-length Hollerith constant**
	An illegal Hollerith constant of length 0 was used in the program.
F2035	**Hollerith constant : text length disagrees with given length**
	A Hollerith constant was smaller than the size given in the length field of the Hollerith constant.

Number	Compiler Compilation Error Message
F2036	*character* **: non-FORTRAN character**
	A special character in the source file was not recognized.
F2037	**illegal label field**
	A nondigit value was used in a label field.
F2038	**zero-value label field**
	A label with the value 0 was used in the program.
	Labels must have values between 1 and 99,999, inclusive.
F2039	**free-form label too long**
	A label was more than five digits long.
F2040	**label on continuation line**
	A label was declared on a continuation line.
F2041	**first statement line must have ' ' or '0' in column 6**
	A continuation character was used on the first statement line in the program.
F2042	**label on blank line**
	A label was used on a line with no statements.
F2043	**alternate bases illegal**
	Alternate integer bases are not allowed if the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect).
F2044	**" nonstandard character string delimiter**
	The quotation mark is a nonstandard Microsoft extension character-string delimiter.
F2050	**$DEFINE :** *name* **already defined**
	A name may only be given a **$DEFINE** value once. To re-define, **$UNDEF** the variable first.
F2051	**invalid expression in metacommand**
	The compiler could not interpret an expression in an **$IF**, **$ELSE**, **$ELSEIF**, **$ENDIF**, or **$DEFINE** metacommand.

Number	Compiler Compilation Error Message

F2052 **unmatched parenthesis**

A closing parenthesis was found without an opening parenthesis preceding it.

F2053 *character* **: unrecognized character**

The compiler encountered an operator it did not recognize or a name beginning with a nonalphabetic character.

F2054 *name* **: not defined**

A name was referenced which had not been **$DEFINE**d.

F2055 **logical operator expected**

An **.AND.** or **.OR.** operator was expected: the other operators cannot appear consecutively.

F2056 *character* **: unexpected characters at end of expression**

There were additional characters on the line that could not be part of the expression.

F2059 **operand expected**

The operator requires an operand.

F2060 **invalid integer constant**

The constant is out of range for integer type or not in correct form for a constant.

F2062 **relational operator expected**

The operands supplied must have a relational operator.

F2064 *name*: **defined with no value**

If a name was **$DEFINE**d with no value, it can only be used as an argument to **DEFINE( )**.

F2070 *filename* **: cannot open include file**

The specified include file could not be opened because it was not found in the source directory or in other directories specified by the include search paths given on the command line.

F2071 **INCLUDE : quoted string missing**

The argument for the **INCLUDE** statement or **$INCLUDE** metacommand was not of **CHARACTER** type.

Number	Compiler Compilation Error Message
F2072	**INCLUDE : argument must be character constant**
	The argument must be enclosed in apostrophes or quotes.
F2101	**DO : too many expressions**
	A **DO** statement had more than three items following the equal sign (=).
F2102	**I/O implied-DO list : list empty**
	No items appeared in an I/O implied-**DO** list.
F2103	**I/O implied-DO list : too many expressions**
	More than three expressions appeared after the equal sign (=) in an I/O implied-**DO** list.
F2104	**I/O implied-DO list : illegal assignment**
	Only one assignment is legal in an I/O implied-**DO** list.
F2105	**I/O implied-DO list : too few expressions**
	Fewer than two expressions followed the equal sign (=) in an I/O implied-**DO** list.
F2106	**I/O implied-DO list : assignment missing**
	No assignment appeared in an I/O implied-**DO** list, or more than two expressions in the list were embedded in parentheses.
F2107	**assignments in COMPLEX constant illegal**
	An illegal embedded assignment appeared in a constant of type **COMPLEX**.
F2108	**illegal assignment in parenthesized expression**
	An illegal embedded assignment appeared in an expression enclosed in parentheses.
F2111	**numeric constant expected**
	A symbolic or numeric constant did not appear as part of a complex constant.
F2112	*name* **: not symbolic constant**
	The specified name was not a symbolic constant.
F2113	**component of COMPLEX number not INTEGER or REAL**
	A component of a **COMPLEX** number was not of type **INTEGER** or **REAL**.

Number	Compiler Compilation Error Message
F2114	**parser stack overflow, statement too complex**
	The statement being parsed was too large for the parser.
F2115	**syntax error**
	The source file contained a syntax error at the specified line.
F2124	**CODE GENERATION ERROR** **Contact Microsoft Technical Support**
	The compiler could not generate code for an expression. Usually this error occurs with a complicated expression.
	Try rearranging the expression. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F2125	*name* **: allocation exceeds 64K**
	The specified item exceeded the limit of 64K.
	Huge arrays are the only items that are allowed to be larger than 64K.
F2126	*name* **: automatic allocation exceeds 32K**
	The subroutine or function *name* has an exceedingly large amount of compiler-generated temporary variables that take up more than 32,767 bytes.
	Try splitting the subroutine or function into smaller pieces.
F2127	**parameter allocation exceeds 32K**
	The storage space required for the arguments to a function exceeded the limit of 32K.
F2128	*name* **: huge array cannot be aligned to segment boundary**
	The specified array violated one of the restrictions imposed on huge arrays.
F2200	*subprogram* **: formal argument** *name* **:** **CHARACTER*(*) cannot pass by value**
	Arguments that are passed by value must have a length that can be determined at run time. **CHARACTER*(*)** lengths are determined at run time.
F2201	*subprogram* **: type redefined**
	The type given in the specified **ENTRY**, **FUNCTION**, or **SUBROUTINE** statement was redefined. It was defined with a different type in an earlier subprogram.

Number	Compiler Compilation Error Message
F2202	*subprogram* **: defined with different number of arguments** The specified **ENTRY**, **FUNCTION**, or **SUBROUTINE** statement was defined or used earlier in the program with a different number of arguments.
F2203	*subprogram* **: formal argument** *name* **: symbol-class mismatch** The specified formal argument was defined previously with a different class. An **EXTERNAL** statement that passes a function to a variable, or a similar symbol-class mismatch, can cause this error.
F2206	**ENTRY seen before FUNCTION or SUBROUTINE** An **ENTRY** statement appeared before any **FUNCTION** or **SUBROUTINE** statements in the program. An **ENTRY** statement can only appear in functions and subroutines.
F2207	**ENTRY not in function or subroutine** An **ENTRY** statement appeared in a subprogram that was not a function or subroutine. It may have appeared in the main program.
F2208	*name* **: formal argument used as ENTRY** The specified name was used as a formal argument in an earlier **ENTRY** statement or in the subprogram header in the current subprogram.
F2209	*name* **: illegal as formal argument** The symbol class of the formal argument was illegal. A formal argument can only be a variable, array, subroutine, function, or entry point.
F2210	*name* **: formal argument redefined** The specified formal argument appeared in the argument list more than once.
F2211	**alternate RETURN only legal within subroutine** An alternate **RETURN** statement was specified outside of a subroutine.
F2212	*subprogram* **: subprogram used or declared before INTERFACE** The specified subprogram was used or declared before the corresponding **INTERFACE** statement appeared in the program.
F2213	*subprogram* **: already defined** The specified subprogram was already defined in the current module.

Number	Compiler Compilation Error Message
F2214	*subprogram* : **already used or declared with different symbol class**

The specified subprogram was used earlier in the program with a different class. For example, a subprogram that was used earlier in the program as a function and then declared as a subroutine would cause this error.

F2215 *subprogram* : **ENTRY : CHARACTER lengths differ**

In a subprogram, if an entry name of type **CHARACTER** is used, then all of the entry names in that subprogram must be of type **CHARACTER**. If one entry name is of type **CHARACTER*(*)**, then all must be of that type.

F2216 *subprogram* : **CHARACTER and non-CHARACTER types mixed in ENTRY statements**

CHARACTER and non-**CHARACTER** types were mixed in a subprogram.

F2217 **too many PROGRAM statements**

More than one **PROGRAM** statement appeared in the source file.

Only one **PROGRAM** statement is allowed per program.

F2218 *name* : **used or declared before ENTRY statement**

The name in an **ENTRY** statement was declared or used previously in the same subprogram. This caused a symbol-class conflict that prevented the name from being used in an **ENTRY** statement.

F2219 *subprogram* : **formal argument** *name* : **VALUE/REFERENCE mismatch**

An **INTERFACE** statement or prior call specified a different way of passing this argument than that specified in the current declaration.

F2220 *subprogram* : **length redefined**

The length of a function when called was different than when it was defined.

F2221 *subprogram* : **formal argument** *name* : **NEAR/FAR/HUGE mismatch**

The **NEAR**, **FAR**, or **HUGE** attributes were defined differently in the **INTERFACE** statement than in the subprogram definition or its arguments.

F2222 *name* : **formal argument previously initialized**

The formal argument to an **ENTRY** statement appeared previously in a **DATA** statement within the same subprogram.

Number	Compiler Compilation Error Message
F2223	*subprogram* : **formal argument** *name* : **subprogram passed by VALUE** The formal argument had the **VALUE** attribute. Subprograms cannot be passed to items with the **VALUE** attribute.
F2224	*name* : **language attribute mismatch** Language attributes were declared differently in the subprogram declaration than in the **INTERFACE** statement.
F2225	*name* : **NEAR/FAR attribute mismatch** The **NEAR** or **FAR** attribute was used differently in the **INTERFACE** statement than in the subprogram declaration.
F2226	*name* : **VARYING attribute mismatch** The **VARYING** attribute was not used in both the **INTERFACE** statement and the subprogram declaration.
F2227	*subprogram* : **formal argument** *name* : **previously passed by value, now by reference** A formal argument previously passed by value was passed by reference. The **VALUE** attribute should be specified for the formal argument.
F2228	*subprogram* : **formal argument** *name* : **previously passed by reference, now by value** A formal argument previously passed by reference was passed by value. The **REFERENCE** attribute should be specified for the formal argument.
F2229	*subprogram* : **formal argument** *name* : **previously passed with NEAR, now with FAR or HUGE** An address-length mismatch occurred. This is because an **INTERFACE** statement specifying the **FAR** or **HUGE** attribute for the formal argument was not given.
F2230	*subprogram* : **formal argument** *name* : **previously passed with FAR or HUGE, now with NEAR** An **INTERFACE** statement specifying the **NEAR** attribute for the formal argument was not given.
F2231	*name* : **PROGRAM : name redefined** The program name already exists as a global entity.

Number	Compiler Compilation Error Message
F2232	*subprogram* : **formal argument** *name* : **Hollerith passed to CHARACTER formal argument.** Hollerith constants may only be passed to formal arguments of type integer, logical, or real.
F2233	*name* : **previously called near** A function that was previously declared or referenced with near addressing was used with a far call.
F2234	*name* : **previously called far** A function that was previously declared or referenced with far addressing was used with a near call.
F2299	*name* : **EQUIVALENCE: structure components illegal** A structure variable may appear in an **EQUIVALENCE** statement, but not in an individual field of a structure.
F2301	*name* : **EQUIVALENCE (or COMMON): formal argument illegal** An item other than a local variable or array, or a variable or an array in a common block, appeared in an **EQUIVALENCE** or **COMMON** statement.
F2302	*name* : **EQUIVALENCE : not array** In an **EQUIVALENCE** statement, an item that was not an array had an argument or subscript list attached to it.
F2303	*name* : **EQUIVALENCE : array subscripts missing** A construct such as `x( )` was used to declare the specified array. If no bounds are required, delete the parentheses.
F2304	*name* : **EQUIVALENCE : nonconstant offset illegal** A nonconstant offset was used for an array in an **EQUIVALENCE** statement.
F2305	*name* : **nonconstant lower substring expression illegal** The lower bound of a substring expression was not a constant in an **EQUIVALENCE** statement.
F2306	*name* : **EQUIVALENCE : enclosing class too big** Arithmetic overflow occurred while the offset of an array expression in an **EQUIVALENCE** statement was being calculated.

Number	Compiler Compilation Error Message
F2307	*name* **:** *stmt* **: allocatable array illegal** Allocatable arrays may not appear in **COMMON** or **EQUIVALENCE** lists.
F2308	*name* **: COMMON (or NAMELIST): length specification illegal** It is illegal to specify the length of a type in a **COMMON** statement. Use a separate type statement to declare the length.
F2309	*name* **: COMMON (or NAMELIST): attributes on items illegal** **STRUCTURE** may appear as well as **COMMON**. Fields in a **STRUCTURE** may not have attributes.
F2310	*name* **: COMMON (or EQUIVALENCE) : SUBROUTINE (or FUNCTION) name illegal** A function or subroutine name was included in a **COMMON** or **EQUIVALENCE** statement. Only local variables and arrays are legal.
F2311	*name* **: COMMON (or EQUIVALENCE or STRUCTURE) : preinitialization illegal** Items in **COMMON** or **EQUIVALENCE** statements cannot be preinitialized in type-declaration statements. Use the standard notation for the **DATA** statement.
F2312	*name* **: COMMON (or EQUIVALENCE or NAMELIST) : formal argument illegal** The specified formal argument was used in a **COMMON** or **EQUIVALENCE** statement.
F2313	*name* **: COMMON (or EQUIVALENCE or NAMELIST) : not an array or variable** An item other than an array or variable was used in an **EQUIVALENCE** or **COMMON** statement.
F2314	*array* **: COMMON : too big** Arithmetic overflow occurred while the size of a common block was being calculated. **STRUCTURE** can also appear. Common blocks can be very large; structures must be less than 64K bytes in size.
F2315	*array* **: COMMON : array size nonconstant or zero** A nonconstant or 0 value was used to dimension the array.

Number	Compiler Compilation Error Message
F2316	*name1*, *name2* : **EQUIVALENCE : both in blank common block**

Two items specified in an **EQUIVALENCE** statement at different offsets were both in a blank common block. In the **EQUIVALENCE** statement, these items were specified to be at the same location in memory.

F2317 *name1*, *name2* : **EQUIVALENCE : both in common block** *commonblock*

Two items specified in an **EQUIVALENCE** statement at different offsets were both in a named common block. These items were specified in the **EQUIVALENCE** statement to be at the same location in memory.

F2318 *name1*, *name2* : **EQUIVALENCE : in different common blocks**

Two items in different common blocks were specified in an **EQUIVALENCE** statement.

F2319 *name* : **EQUIVALENCE : extends blank common block forward**

In an **EQUIVALENCE** statement, it is illegal to increase the size of a blank common block by adding memory elements before the beginning common block declared in the **COMMON** statement.

F2320 *name* : **EQUIVALENCE : extends common block** *commonblock* **forward**

In an **EQUIVALENCE** statement, it is illegal to increase the size of a named common block by adding memory elements before the beginning common block declared in the **COMMON** statement.

F2321 *name1*, *name2* : **EQUIVALENCE : conflicting offsets**

The processing of an **EQUIVALENCE** statement detected two items that should have had the same offsets but did not. Inconsistent use of **EQUIVALENCE** statements caused this problem.

F2322 *name* : **EQUIVALENCE : two different common blocks**

An **EQUIVALENCE** statement placed an item in two different common blocks.

F2323 *commonblock* : **COMMON : size changed**

The size of the specified common block differed from the size allocated in a prior subprogram.

F2324 *commonblock* : **COMMON : too big to be NEAR**

The specified common block, declared with the **NEAR** attribute, is larger than a segment.

Number	Compiler Compilation Error Message
F2325	*name* : **COMMON : function or subroutine name** The specified name was used as both a common-block name and a function or subroutine name.
F2326	*name* : **already in COMMON** The specified name appeared in a **COMMON** statement elsewhere in this subprogram.
F2327	*name* : **EQUIVALENCE : needs at least two items** An **EQUIVALENCE** statement had fewer than two items in a class.
F2328	*name* : **already typed** The specified item appeared in an earlier type statement in the same subprogram or derived-type definition.
F2329	**blank common cannot be HUGE** In medium model, blank common items must be smaller than a single segment. Named common items do not have this restriction.
F2330	*name* : **already dimensioned** Array bounds appeared for the specified item in an earlier specification statement in the same subprogram.
F2331	*name* : **types illegal on BLOCK DATA/COMMON/PROGRAM/SUBROUTINE** The specified item was not one of the symbol classes that can be typed.
F2332	*name* : **cannot initialize in type statements** An attempt was made to initialize the specified item in a type statement while the /4Ys compiler option was used or the **$STRICT** metacommand was in effect.
F2333	*name* : **DIMENSION : not array** The specified item in a **DIMENSION** statement (for example, an item already declared in an **EXTERNAL** or **PARAMETER** statement) was not an array.
F2334	*name* : **ALLOCATABLE: bounds must be omitted** An allocatable array declaration's subscripts must be ':' only, with no upper or lower parameters.

Number	Compiler Compilation Error Message
F2336	*array* : **array bounds missing**

Both bound expressions were missing from the declaration of the specified array.

At least an upper bound must be present.

F2337 *array* : * : **not last array bound**

An assumed-size array was declared with an asterisk (*) that did not occur in the last bound.

F2338 *array* : **bound size too small**

The bound size of the specified array was not a positive whole number.

Bounds of adjustable-size arrays can be checked at run time. This compile-time error occurs only when the upper and (possibly implicit) lower bounds create a negative or zero element count for an array bound.

F2339 *array* : **adjustable-size array not in subprogram**

The specified adjustable-size array was declared in a subprogram declared with a **PROGRAM** or **BLOCK DATA** statement.

An adjustable-size array is legal only in an **ENTRY, FUNCTION,** or **SUBROUTINE** statement in a subprogram.

F2340 **IMPLICIT already seen**

An **IMPLICIT** statement has already been seen, so an **IMPLICIT NONE** statement is illegal.

F2341 *letters* : **IMPLICIT : only single letter allowed**

The upper or lower value of the range in an **IMPLICIT** statement was not a single character.

F2342 **IMPLICIT NONE already seen**

There may only be one **IMPLICIT NONE** statement per subprogram.

F2343 *letter1*, *letter2* : **IMPLICIT : lower limit exceeds upper limit**

The upper letter in the range in an **IMPLICIT** statement had a smaller value than the letter in the lower range.

F2344 *letter* : **already IMPLICIT**

The specified character already appeared in an **IMPLICIT** statement earlier in the same subprogram.

Number	Compiler Compilation Error Message
F2345	*name* **: illegal use of SAVE (or AUTOMATIC, EXTERNAL, INTRINSIC, PARAMETER)** The specified *name* appeared earlier in a conflicting type statement.
F2346	*name* **: INTRINSIC : unknown name** The specified *name* is not the name of a supported intrinsic function.
F2348	*name* **: already declared SAVE (or AUTOMATIC, EXTERNAL, INTRINSIC, PARAMETER)** The specified *name* was declared more than once with the same type of statement.
F2349	*name* **: PARAMETER : nonconstant expression** The specified item was declared with a nonconstant value in a **PARAMETER** statement.
F2350	**RECORD : structure type illegal in IMPLICIT statement** Structure variables cannot be implicitly typed.
F2351	*name* **: repeated in formal-argument list** The specified item was repeated in the formal-argument list to a statement function.
F2352	*name* **: formal argument not local variable** Only local variables can be used as formal arguments to statement functions.
F2354	*name* **: statement function already declared** The specified statement function was already declared in the current subprogram.
F2355	*name* **: formal argument not a variable** An argument list or substring operator for the specified item appeared in the formal-argument list to a statement function.
F2356	*name* **: statement function : too few actual arguments** More formal arguments than actual arguments were declared for a statement function.
F2357	*name* **: statement function : too many actual arguments** More actual arguments than formal arguments were declared for a statement function.

Number	Compiler Compilation Error Message
F2359	*type* **: illegal length**

An illegal length specifier for the given type was used in a declaration. For example, `REAL*13` would cause this error.

F2360 **no matching [END] STRUCTURE/UNION/MAP statement**

A **STRUCTURE**, **UNION**, or **MAP** statement was seen without a matching **END STRUCTURE /UNION/MAP** statement, or an **END** statement was seen without a matching **STRUCTURE/UNION/MAP** statement.

F2361 *stmt* **: not a name**

The argument on the **STRUCTURE** or **END STRUCTURE** statement was not a symbol name.

F2362 **integer constant expression expected**

An integer value or integer constant expression was expected for an optional type-length specification.

F2363 **length** *value* **: illegal type length**

A zero or negative length specifier was used in a type statement, or the length specifier was larger than the largest allowed for all types.

F2364 **only C attribute legal on INTEGER type**

An attribute other than the C attribute appeared with an **INTEGER** type statement.

F2365 **attributes illegal on non-INTEGER types**

Attributes in type statements are illegal, other than the C attribute on the **INTEGER** statement. Attributes were not put on the variables themselves.

F2366 **DOUBLE PRECISION : length specifier illegal**
DOUBLE COMPLEX : length specifier illegal

A **DOUBLE PRECISION** or **DOUBLE COMPLEX** statement included a length specifier. **DOUBLE PRECISION** is the same as **REAL*8**. **DOUBLE COMPLEX** is the same as **COMPLEX*8**.

F2367 **value** *value* **: INTEGER : range error**

The specified constant was out of range for type conversion, or the type of an integer item was in conflict with the integer size specified in the /4I compiler option (or **$STORAGE** metacommand). For example, the following code induces this error:

Number	Compiler Compilation Error Message
F2367	*(continued)*

```
$STORAGE:2
      INTEGER*4 i
      i = 300000+30000
      i =     10* 4000
      = -30000-30000
      END
```

To correct this error, use the appropriate **INT2** or **INT4** intrinsic function to make sure the appropriate (2- or 4-byte) arithmetic is performed on the variable.

F2368 *name* **: truncated to 6 characters**

When the /4Ys compiler option is used or the **$STRICT** metacommand is in effect, only six characters can appear in identifier names.

F2369 *name* **: $ illegal in C name**

A character in the specified name was illegal for a C variable. C variables allow only underscores (_) and alphanumeric characters in names.

F2370 **length specification illegal**

When the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), length specifications can only be used with **CHARACTER** type statements.

F2371 *name1*, *name2* **: EQUIVALENCE : character and noncharacter items mixed**

Character and noncharacter items were mixed in an **EQUIVALENCE** statement.

F2372 *name* **: more than 7 array bounds**

When the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), an array cannot have more than seven bounds.

F2373 *name* **: REFERENCE or VALUE only legal on formal arguments**

A **REFERENCE** or **VALUE** attribute was used with an item that was not declared in the formal-argument list for the routine.

If the item is used in a type statement, then the attributed item must also appear in the formal-argument list of a subprogram. If the item appears in an **ENTRY** statement, include the attribute there instead.

F2374 *name* **: attributes illegal on array bounds**

No attributes are allowed on items that are used when dimensioning arrays.

Number	Compiler Compilation Error Message
F2375	*name* **: assumed-size array : cannot pass by value**

An assumed-size array was passed as an actual argument to a routine that had its formal argument declared with the **VALUE** attribute.

F2376 *name* **: adjustable-size array : cannot pass by value**

An adjustable-size array was passed as an actual argument to a routine that had its formal argument declared with the **VALUE** attribute.

F2377 *name* **: NEAR common block has HUGE item**

A common block declared with the **NEAR** attribute included item(s) that required the common block to be huge.

F2378 *name* **: NEAR array bigger than segment**

An array declared with the **NEAR** attribute was larger than a segment.

F2379 *name* **: item in common block crosses segment**

An item or an array element in a common block crossed a segment boundary. Items or arrays must be evenly aligned to signal boundaries when a common block crosses a segment.

F2380 *name* **: VARYING illegal on symbol class**

The **VARYING** attribute was used on something other than a function or subroutine.

F2381 *commonblock* **: NEAR/FAR/HUGE attribute mismatches default**

An attribute declared for the given common block was different from the attribute implicitly applied to the common block in an earlier subprogram.

In medium-model programs, the **NEAR** attribute is used implicitly, unless the size of the common block requires the common block to be huge. In large-model programs, the **FAR** or **HUGE** attribute is used implicitly.

F2382 *commonblock* **:** *attribute* **attribute mismatch with earlier NEAR/FAR/HUGE**

An attribute given in an earlier common-block declaration (possibly in a different subprogram) was not the same as the current attribute.

F2383 *name* **: COMMON : character and noncharacter items mixed**

Character and noncharacter items cannot be mixed in a common block when the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect).

Number	Compiler Compilation Error Message
F2384	*name* : *attribute* **variable cannot be AUTOMATIC** The attribute can be one of the following four: **HUGE, FAR, EXTERN, ALLOCATABLE.**
F2385	*name* : **STRUCTURE has no elements** A **STRUCTURE** was declared with no component fields.
F2386	**NAMELIST : group name required** A name must appear between the slashes.
F2387	*name* : **STRUCTURE : intrinsic type name** Derived type names may not be the same as the names of intrinsic types.
F2388	*name* : **NAMELIST : array bounds illegal** Only variable and array names are allowed in a **NAMELIST** statement (the array must be declared someplace else).
F2389	*name* : **not a STRUCTURE name** The argument in the **STRUCTURE( )** statement has not previously been declared in a **STRUCTURE** statement.
F2390	*name* : **ALLOCATABLE : common block variable illegal** Common block variables may not be declared with the allocatable attribute.
F2391	*name* : **ALLOCATABLE : equivalenced variable illegal** Equivalenced variables may not be given the allocatable attribute.
F2392	**MAP : no enclosing UNION statement** A map can only appear within a **UNION** statement.
F2393	*name* : **element name conflicts with operator**
F2394	*name* : **NAMELIST: structure illegal** Structure variables cannot appear in **NAMELIST** groups.
F2395	**UNION : not in a STRUCTURE** A union can only appear within a **STRUCTURE** statement.
F2396	*name* : **ALLOCATABLE : must be in array** A variable was given the allocatable attribute but not dimensioned.

Number	Compiler Compilation Error Message
F2401	*name* : **DATA : illegal address expression**
	An illegal expression was used for the offset in a **DATA** statement.
	Only constant offsets are legal for items in **DATA** statements.
F2402	*name* : **cannot initialize formal argument**
	The item being initialized was a formal argument to a subprogram.
F2403	*name* : **cannot initialize item in blank common block**
	An attempt was made to use a **DATA** statement to initialize the specified item in a blank common block.
F2404	*name* : **can only initialize common block in BLOCK DATA subprogram**
	An attempt was made to initialize the specified item named in a common block in a **DATA** statement.
	Items in named common blocks can be initialized only in **BLOCK DATA** subprograms.
F2405	*name* : **DATA : not an array or variable**
	Only arrays and variables can be initialized in **DATA** statements.
F2406	*name* : **repeat count not positive integer**
	The repeat count for initialization of the specified item was not a positive integer value.
F2407	*name* : **DATA : nonconstant item in initializer list**
	A nonconstant value was used to initialize the specified item in a **DATA** statement.
F2408	*name* : **DATA : too few constants to initialize item**
	The **DATA** statement did not include enough constants to initialize the specified item.
F2409	*name* : **nonstatic address illegal in initialization**
	During processing of an implied-**DO** list in a **DATA** statement, the specified item did not have a static address.
	Make sure the item has constant subscript specifiers.

Number	Compiler Compilation Error Message
F2410	*name* **: bound or increment not constant**

The specified item in implied-**DO** initialization in a **DATA** statement did not have constant bounds.

F2411 *name* **: DATA : zero increment**

In a **DATA** statement, the increment value in the implied-**DO** list for the specified item must be set so the loop exits.

F2412 *name* **: DATA : active implied-DO variable**

The specified implied-**DO** variable was used in nested **DATA** implied-**DO** initialization loops in a **DATA** statement.

F2413 *name* **: DATA : implied-DO variable not INTEGER**

In **DATA** statements, only implied-**DO** variables of types **INTEGER** and **INTEGER***n* are legal.

F2414 *name* **: DATA : not array-element name**

Only array elements can be initialized in implied-**DO** initializations in **DATA** statements.

F2415 **DATA : too few constants to initialize names**

The constant list was exhausted before the initialization of the name list was complete.

F2416 *name* **: bound or increment not INTEGER**

When the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), only items of type **INTEGER** are allowed for **DATA** implied-**DO** loop bounds and increments. Otherwise, any arithmetic type is allowed and is truncated to type **INTEGER** by an implicit use of the **INT** intrinsic function.

F2417 **DATA : iteration count not positive**

In the implied-**DO** list (*...,dovar=start,stop,inc)*, if the increment *inc* is positive, then *start* must be greater than *stop*; if the increment *inc* is negative, then *stop* must be greater than *start*. If not, then the loop would execute zero times, which is not allowed.

F2418 *name* **: variable was declared automatic**

A variable which is going to go on the stack (declared **AUTOMATIC**) cannot be given the **HUGE** attribute.

Number	Compiler Compilation Error Message
F2419	*name* **: cannot initialize allocatable array**
	An allocatable array cannot appear in a **DATA** statement.
F2420	*arrayname* **: ALLOCATABLE: dummy argument illegal**
	The dummy argument, *arrayname*, was declared as an allocatable array.
	Dummy arguments may not be declared as allocatable arrays. Change the declaration of the dummy variable, for example, to static.
F2490	*name* **: STAT= must be last parameter**
	The **STAT=** parameter must be last in an **ALLOCATE** statement.
F2491	*name* **:** *stmt* **: not allocatable array**
	The name in the allocate or deallocate statement is not an allocatable array. (The second parameter is **ALLOCATE** or **DEALLOCATE**.)
F2492	*stmt* **: STAT= variable must be scalar integer**
	First parameter is **ALLOCATE** or **DEALLOCATE**.
F2493	*name* **: * : illegal bound**
	There are no assumed-size dimensions in an allocatable array.
F2500	*array* **: adjustable-size array : used before definition**
	An adjustable-size array was used before it was seen in an **ENTRY** statement.
F2502	*type* **: cannot convert to** *type*
	When the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), constants cannot be converted between **CHARACTER** and nonCHARACTER types.
F2503	*intrinsic* **: incorrect use of intrinsic function**
	Invalid arguments were given for the specified intrinsic function.
F2504	*intrinsic* **: multiple arguments**
	The specified intrinsic function had more than one argument; only one is legal.
F2505	*intrinsic* **: cannot convert FAR address to NEAR**
	An item in the specified intrinsic function can only be referenced with addressing consistent with the **FAR** or **HUGE** attribute.

Number	Compiler Compilation Error Message
F2506	**cannot convert type to** *type* An invalid type conversion to **CHARACTER** or **LOGICAL** type was attempted.
F2508	*array* **: array bound used array reference** An expression having an array was used when declaring an adjustable-size array. Only simple variables in common blocks on the current subprogram's formal argument list are allowed as variables in the bound expression.
F2509	*element* **: not an element of** *name* The name given as a component of a derived type is undefined.
F2510	*name* **: symbolic constant : subscript illegal** The specified symbolic constant had an array index or argument list.
F2511	*name* **: symbolic constant : substring illegal** The specified symbolic constant had a substring operator.
F2512	*name* **: variable : argument list illegal** The specified simple variable included an argument list.
F2513	*name* **: not a variable** The specified item was not a variable. A variable is expected in this context.
F2514	**concatenation with CHARACTER*(*)** A **CHARACTER*(*)** item was used in a concatenation operation. Only items with specified lengths are legal in concatenations.
F2515	**left side of assignment illegal** The left side of an assignment statement was illegal. Only variables, array elements, or function-return variables may appear on the left side of assignment statements.
F2516	*name* **: assignment using active DO variable illegal** An active **DO** variable was used in an assignment statement.
F2517	**illegal implied-DO list in expression** In this context, implied-**DO** statements are illegal in expressions.

Number	Compiler Compilation Error Message
F2518	*name* : **not a structure**
	A variable which is not of derived type was specified with a component field.
F2519	*name* : **operation error with COMPLEX operands**
	A constant-folding error occurred. The number created would probably overflow the allowed storage. Using smaller numbers will correct.
F2520	*name* : **operation error with REAL operands**
	A constant-folding error occurred. The number created would probably overflow the allowed storage. Using smaller numbers will correct.
F2521	**negative exponent with zero base**
	A negative exponent was used with a zero-value base.
F2522	**division by zero**
	Division by zero occurred during constant folding.
F2523	**only comparisons by .EQ. and .NE. allowed for complex items**
	Only **.EQ.** and **.NE.** are legal as comparison operators for complex items.
F2524	**non-numeric operand**
	A nonarithmetic operand was specified with an arithmetic operator.
F2525	**exponentiation of COMPLEX and DOUBLE PRECISION together illegal**
	When the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), exponentiation is illegal with bases of type **COMPLEX** having **DOUBLE PRECISION** exponents, or with bases of type **DOUBLE PRECISION** having **COMPLEX** exponents.
F2526	**concatenation of expressions illegal**
	An illegal concatenation operation occurred. If a noncharacter item is used in a concatenation, it must be a constant or it must be addressable.
F2527	**noncharacter operand**
	When the /4Ys compiler option is used (or when the **$STRICT** metacommand is in effect), concatenation operators can be used only with character operands.

Number	Compiler Compilation Error Message
F2528	**nonlogical operand**

Logical operators (**.AND.**, **.OR.**, **.NOT.**, **.EQV.**, and **.NEQV.**) must be used with logical operands.

F2529 **operands of relation not numeric or character**

Relational operators (**.LT.**, **.LE.**, **.GT.**, **.GE.**, **.EQ.**, and **.NE.**) must be used with arithmetic or character operands.

F2530 *name* **: symbol class illegal here**

The class of the given symbol was illegal in this context.

F2531 *name* **: bound not integer**

The /4Ys compiler option was used in compiling (or the **$STRICT** metacommand was in effect), and a substring had a noninteger substring-bound expression.

If the /4Ns compiler option is used in compiling (or the **$NOTSTRICT** metacommand is in effect), any arithmetic expression is legal and is truncated to integers through an implicit use of the **INT** intrinsic function.

F2532 *name* **: substring on noncharacter item**

An attempt was made to take the substring from an item that was not a character variable or array item.

F2533 *name* **: lower substring bound exceeds upper bound**

The /4Yb compiler option was used (or the **$DEBUG** metacommand was in effect), and the value of the upper substring bound was less than the value of the lower substring bound.

F2534 *name* **: upper substring bound exceeds string length**

The /4Yb compiler option was used (or the **$DEBUG** metacommand was in effect), and the upper substring bound was greater than the length of the item from which the substring was taken.

This error occurs only if the length of the item was not specified (that is, if it was declared as a **CHARACTER***n* item).

F2535 *name* **: lower substring bound not positive**

The lower substring bound was less than or equal to 0.

The minimum value for items of type **CHARACTER** is 1.

Number	Compiler Compilation Error Message
F2536	*array* : **subscript** *number* **out of range** The /4Yb compiler option was used (or the **$DEBUG** metacommand was in effect), and a local array or an array in a common block had a bound out of range.
F2537	*array* : **array subscripts missing** The specified array, which did not have array subscripts, was used in an expression.
F2538	*array* : **subscript** *number* : **not integer** When the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), a subscripting expression used in the specified array must be of type **INTEGER**. Otherwise, it must be an arithmetic type that is truncated to **INTEGER** by an implicit use of the **INT** intrinsic function.
F2539	*array* : **too few array subscripts** Not enough subscripts were given when the specified array was used in an expression.
F2540	*array* : **too many array subscripts** Too many subscripts were given when the array was used in an expression.
F2541	**cannot convert between CHARACTER and non CHARACTER constants** If the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), constants cannot be converted between **CHARACTER** and non**CHARACTER** types.
F2542	**one numeric, one character operand** If the /4Ys compiler option is used in compiling (or the **$STRICT** metacommand is in effect), both operands used with relational operators must be character or both must be arithmetic. Operands of different types cannot be mixed.
F2543	**operand type must be logical or integer** See F2528. When **the $STRICT** metacommand is off, integer operands are allowed also.
F2544	**operand types do not match** One argument is a derived type and the other argument is not.
F2545	**invalid operator for structure operands** Only the **.EQ.** and **.NE.** operators are defined for structures.

Number	Compiler Compilation Error Message
F2559	*array* **: array bound used illegal variable**

Only variables in common blocks or in the formal argument list to the current subprogram are legal when declaring adjustable-size arrays.

F2560 *array* **: array bound used intrinsic call**

Only variables in common blocks or in the formal argument list to the current subprogram are legal when declaring adjustable-size arrays.

F2561 *array* **: array bound used function call**

Only variables in common blocks or in the formal argument list to the current subprogram are legal when declaring adjustable-size arrays.

F2562 **cannot pass CHARACTER*(*) by value**

The program tried to pass by value an item of type **CHARACTER*(*)**. This is illegal because the length of such items is not known at compile time.

Actual arguments with a length of *n* can be passed to **CHARACTER*n** items, and these items can be passed by value, if required.

F2563 **incompatible types for formal and actual arguments**

The types of the formal and actual arguments did not match.

Formal and actual arguments must have the same types (except for arguments of type **CHARACTER**, where the lengths can differ).

F2564 **incompatible types in assignment**

The expressions on the left and right sides of an assignment statement were of different types. For example, a logical expression cannot be assigned to an integer variable.

F2565 *operation* **: COMPLEX : type conversion error**

An attempt was made to convert values of one type to types that hold a smaller range of values.

F2566 *operation* **: REAL : type conversion error**

An attempt was made to convert values of one type to types that hold a smaller range of values.

F2567 **LEN : illegal expression**

Only constants, symbols, concatenations, intrinsic type casts, and strings are allowed in the **LEN** intrinsic function.

Number	Compiler Compilation Error Message
F2568	*name* **: illegal bound type**
	Only integer items are allowed as array bounds when the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect). Otherwise, arithmetic types are allowed and are converted through an implicit use of the **INT** intrinsic function.
F2569	*name* **: Hollerith constant passed by value**
	A Hollerith constant must be passed by reference to a logical, real, or integer formal argument.
F2570	**consecutive arithmetic operators illegal**
	Unary plus and minus cannot follow other arithmetic operators. For example, `I = I**-1` is illegal; `I = I**(-1)` must be used instead.
F2571	**consecutive relational operators illegal**
	The **.NOT.** operator cannot follow another **.NOT.** operator.
F2572	**illegal use of Hollerith constant**
	Hollerith constants are only allowed in assignments, **DATA** statements, and subprogram references.
F2600	*name* **: directly recursive**
	A subprogram is trying to call itself.
F2601	*intrinsic* **: intrinsic function illegal as actual argument**
	This intrinsic function is illegal as an actual argument. (Some specific versions of the generic intrinsic functions can be passed as actual arguments.)
F2602	*name* **: formal argument** *name* **: character expressions cannot be passed by VALUE**
	A character expression cannot be passed by value.
F2604	*subprogram* **: function : argument list missing**
	The specified function was missing an argument list. At least an empty argument list (( )) must be present in expressions.
F2605	*subprogram* **: function : substring operator illegal**
	A substring operator was used illegally with the specified routine name. Substring operators can only be used with arrays and variables.

Number	Compiler Compilation Error Message
F2606	*subprogram* **: formal argument** *name* **: type mismatch** The type of a formal argument was different from the type of the actual argument used in the subprogram call.
F2607	*subprogram* **: formal argument** *name* **: length mismatch** The length of a formal argument was different from the length of the actual argument used in the subprogram call.
F2608	*subprogram* **: formal argument** *name* **: Hollerith illegal with CHARACTER** Hollerith constants can only be used with items of type **INTEGER**, **LOGICAL**, and **REAL** in **DATA** statements and subprogram references.
F2609	*subprogram* **: formal argument * : actual not alternate-return label** Because the specified formal argument was an alternate-return label, the current argument must also be an alternate-return label.
F2610	*subprogram* **: formal argument** *name* **: not alternate-return label** Because the specified formal argument was not an alternate-return label, the current argument must not be an alternate-return label.
F2611	*subprogram* **: formal argument** *name* **: actual not subprogram** The formal argument used in a subprogram declaration was a subprogram, but the actual argument was not.
F2612	*subprogram* **: NEAR formal argument** *name* **: actual has FAR or HUGE address** It is illegal to pass an item that must be addressed with far or huge addressing to a formal argument that must be addressed with near addressing.
F2615	*name* **: not function or subroutine** The specified item was not a function or subroutine. Check the item's use or declaration earlier in the program.
F2616	*subprogram* **: illegal use of function or subroutine** The program tried to use a function as a subroutine or use a subroutine as a function.
F2617	*subprogram* **: adjustable-size array** *array* **: cannot pass by value** An attempt was made to pass an adjustable-size array by value.

Number	Compiler Compilation Error Message
F2618	*subprogram* : **cannot use CHARACTER*(*) function** **CHARACTER*(*)** functions cannot be directly referenced. They can only be passed as actual arguments.
F2619	*name* : **value argument bigger than segment** An argument with a **VALUE** attribute was too big to be passed onto the stack.
F2620	*subprogram* : **formal argument** *name* : **subprogram mismatch** The type of the formal argument to the subprogram was not the same as the actual argument. Both the formal and the actual argument must be subroutines or functions.
F2621	*name* : **formal argument** *name* : **not subprogram** The actual argument to the subprogram was a subprogram, but the formal argument was not a subprogram.
F2622	**assumed-size array** *array* : **cannot pass by value** An assumed-size array can only be passed by reference.
F2623	*name* : **nonconstant CHARACTER length : cannot pass by value** If a substring is used when passing a **CHARACTER*(*n*)** or **CHARACTER*(*)** argument to a formal argument declared with the **VALUE** attribute, then the lower and upper substring values must be constant. Otherwise, the length cannot be determined.
F2624	*subprogram* : **too few actual arguments** The number of actual and formal arguments for the given subprogram did not match. This practice is legal only when the **C** and **VARYING** attributes are specified for the subprogram.
F2625	*subprogram* : **too many actual arguments** The number of actual and formal arguments for the given subprogram did not match. This practice is legal only when the **C** and **VARYING** attributes are specified for the subprogram.

Number	Compiler Compilation Error Message
F2626	**formal argument** *<name>* **: cannot be SAVE or AUTOMATIC** A formal argument cannot appear in either a **SAVE** or **AUTOMATIC** statement.
F2650	*name* **: array expression: cannot be adjustable-size array** An adjustable-size array may not appear in an array expression.
F2651	*name* **: array expression: argument does not conform** The array name is not the same size as other arrays in the expression.
F2652	*name* **: array expression: cannot be assumed-size array** An assumed-size array may not appear in an array expression.
F2653	*<name>* **: array expression : cannot be allocatable array** An allocatable array name cannot appear in an array expression, since its bounds are not known at compile time.
F2702	*iooption* **: array subscript missing** In this context, the array in the specified I/O option cannot appear without subscripts.
F2703	*iooption* **: not** *type* The specified I/O option required an item of a different type. For example, the **REC=***rec* option requires an integer expression.
F2704	*iooption* **: not a variable or array element** The specified I/O option required a variable or an array element, as opposed to an arbitrary expression.
F2705	**label** *number* **: not between 1 and 99999** Statement labels are restricted to the range 1–99,999; they must be one to five digits, not all of which are 0.
F2706	**UNIT= * illegal for this statement** The asterisk (*) unit specifier (console unit) cannot be specified for this I/O statement. Use the asterisk (*) unit specifier only with **READ**, **WRITE**, or **INQUIRE** statements. For **INQUIRE** statements, the asterisk (*) unit specifier is allowed only when the /4Ns compiler option is used (or the **$NOTSTRICT** metacommand is in effect).

Number	Compiler Compilation Error Message

F2707 **illegal unit specifier**

The unit specifier in a **UNIT=** option was not an integer expression, asterisk (*), character variable, array element, array, or substring.

A noncharacter array is a legal unit specifier if the /4Ns compiler option is used in compiling (or the **$NOTSTRICT** metacommand is in effect).

F2708 **illegal format specifier**

The format specifier in a **FMT=** option was not a statement label, integer variable, character expression, character array, noncharacter array, or asterisk (*).

F2709 **HUGE format illegal**

An array declared with a **HUGE** attribute that appeared in a **$LARGE** metacommand, or that spanned more than one segment, was used as a format specifier.

F2710 **UNIT=* : unformatted I/O illegal**

The keyboard or terminal is opened for sequential formatted I/O only.

F2711 **FAR format illegal in medium model**

Data allocated with the **FAR** attribute were used as a format specifier in a medium-model program.

F2712 *iooption* **: appears twice**

The specified I/O option was used more than once in the same I/O statement.

F2713 **END= (or ERR=): illegal statement label**

An integer number in the range 1 to 99,999 must be specified for statement labels following the **ERR=** and **END=** input/output options. (Compile-time error.)

F2714 **I/O option** *number* **: <keyword=> missing**

The I/O option at position *number* in the option list appeared without a keyword.

An I/O option without a keyword must not appear past the second position in the option list. Also, only **UNIT=** and **FMT=** options can appear without a keyword. If the **UNIT=** option appears without a keyword, it must be the first option in the option list. If the **FMT=** option appears without a keyword, it must follow a **UNIT=** option without a keyword.

For example,

```
OPEN (2, 'F.DOT')
```

Number	Compiler Compilation Error Message
F2714	(*continued*) would produce the message `I/O option 2: <keyword=> missing` because the **FILE=** option is missing in the second option.
F2715	*iooption* **: option illegal for this statement** The given I/O option could not be used with this I/O statement.
F2716	**INQUIRE : either UNIT= or FILE= needed** The **INQUIRE** statement must have either a **UNIT=** option or a **FILE=** option, but not both.
F2717	**UNIT= missing** This I/O statement lacked a **UNIT=** option.
F2718	**illegal I/O formatting for internal unit** Internal units do not allow the use of unformatted or list-directed I/O. A format specifier other than asterisk (*) must be used.
F2719	**REC= illegal for internal unit** Direct-access I/O is illegal for internal units.
F2720	**FORMAT : label missing** A **FORMAT** statement lacked a statement label in the range 1 to 99,999.
F2721	**no ASSIGN statements for FMT=<integer variable>** The current format specifier had no corresponding **ASSIGN** statement to set the integer variable to a valid **FORMAT** statement label.
F2722	**UNIT= : not between -32767 and 32767** An external unit number was out of range.
F2723	*iooption* **: unrecognized value in option** An invalid or misspelled value was used with the given I/O option. For example, `ACCESS='DIREKT'` and `ACCESS='RANDOM'` are both illegal.
F2724	**RECL= required to open direct-access file** When opening a file for direct access, the **RECL=** option is required.

Number	Compiler Compilation Error Message

F2725 **illegal input list item**

An input list item was not a variable, array, array element, or substring.

F2726 *iooption* **: * illegal with this option**

The asterisk (*) unit specifier (console unit) cannot be used with the given I/O control specifier.

F2727 *array* **: assumed-size array illegal here**

An assumed-size array cannot be used in this context.

F2728 **attributes are non-standard**

Specifying attributes is nonstandard (-4Ys or **$STRICT** has been specified).

F2729 **FAR or HUGE I/O item illegal in medium model**

Data items having the **FAR** or **HUGE** attribute cannot be used in I/O statements in medium-model programs.

F2730 *name* **: cannot modify active DO variable**

A **DO** variable cannot be modified within its range. For example, the following program fragments cause this error:

```
      DO 100 I = 1,10
      OPEN (33, IOSTAT = I)
  100 CONTINUE

      READ (*,*) (I, I = 1,10)
```

F2731 *iooption* **: noncharacter array nonstandard**

If the /4Ys compiler option is used in compiling (or the **$STRICT** metacommand is in effect), standard forms of the language must be used. In these cases, only character variables, arrays, array elements, and substrings are legal as I/O specifiers.

F2732 *stmt* **: nonstandard statement**

(**IMPLICIT NONE**, **INCLUDE**, **SELECT**, **END DO**, **DO WHILE**, **NAMELIST**, **TYPE**, **END TYPE**, **TYPE ( )**, etc.) The statement is non-standard and **$STRICT** has been specified. **ALLOCATE**, **DEALLOCATE TYPE**, and **INTERFACE** are also nonstandard features.

Number	Compiler Compilation Error Message
F2733	*iooption* **: option nonstandard**

The specified I/O option is not part of standard FORTRAN 77; it cannot be given if the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect).

F2734 **END= : illegal when REC= present**

In **READ** statements, the **REC=** and **END=** options cannot both be present.

F2735 **REC= : illegal when FMT= ***

In **READ** and **WRITE** statements, the **REC=** option is illegal if list-directed I/O is in use.

F2736 **LOCKING : nonstandard**

If the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), the **LOCKING** statement is prohibited.

F2737 *iooption* **: lowercase in string nonstandard**

If the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), the value of the specified I/O option must be given in uppercase. For example, `ACCESS='DIRECT'` is legal in this case, but `ACCESS='direct'` is not.

F2738 *name* **: HUGE internal units illegal**

An array used as an internal unit cannot be declared with the **HUGE** attribute or used in a **$LARGE** metacommand. The array cannot be larger than one segment.

F2739 *name* **: record length too large for internal unit**

For a noncharacter array used as an internal unit, the element size multiplied by the element count (that is, the record length of the internal file) was too large.

F2740 **RECL= : out of range**

The value of the **RECL=** option was less than or equal to 0 or exceeded the maximum legal value.

F2741 **ACCESS= : nonstandard option value**

This option is not allowed for file access.

F2742 **format specification illegal when namelist specified**

The **NML=** and **FMT=**specifiers are mutually exclusive.

Number	Compiler Compilation Error Message
F2743	*name* : **NML= : not a namelist group name**

The name specified by the **NML=** spec was not declared in a **NAMELIST** statement.

F2744 **NML= : namelist group name missing**

The **NML=** spec requires a name as an argument.

F2745 *name* : **i/o of entire structures illegal**

Only structure variable elements can be written to or read from files using formatted I/O.

F2800 *name* : **CHARACTER*(*) type illegal**

An item was declared with **CHARACTER*(*)** type, but it was not in the formal-argument list in the current subprogram.

F2801 **no ASSIGN statements for assigned GOTO (or FMT=)**

The program unit had no **ASSIGN** statements for use with an assigned **GOTO** statement or assigned **FMT=** specifier.

F2803 *name* : **ASSIGN : variable not INTEGER**

Only variables of type **INTEGER*n** or **INTEGER** are legal in **ASSIGN** statements.

F2804 *name* : **ASSIGN : too many INTEGER*1 variables**

Only the first 127 **ASSIGN** statements may use variables of type **INTEGER*1** in a subprogram. This is caused by the storage limitations of **INTEGER*1** items.

F2805 **label** *number* : **redefined in program unit**

The specified label appeared earlier in the subprogram.

Labels may not be defined more than once within a single subprogram unit. This error may also occur if a **DO** loop references a previously defined label.

F2806 **DO-loop variable : not a variable**

A **DO**-loop variable was a symbolic constant, not an actual variable.

F2807 *name* : **illegal use of active DO-loop variable**

It is illegal to use an active **DO**-loop variable as another **DO**-loop variable in a nested **DO** statement. Values cannot be assigned to **DO**-loop variables within **DO** loops.

Number	Compiler Compilation Error Message
F2808	**DO-loop variable not INTEGER or REAL**
	Only variables of type **INTEGER***n* and **REAL***n* are legal as DO-loop variables.
F2809	**DO-loop expression not INTEGER or REAL**
	Only expressions of type **INTEGER***n* and **REAL***n* are legal as **DO**-loop bounds.
F2810	**zero illegal as increment**
	Only nonzero increments are legal as **DO**-loop increments. Otherwise, the loop would never exit.
F2811	**IF or ELSEIF missing**
	No **IF** or **ELSEIF** statement matching an **ELSE** or **ELSEIF** statement appeared in the program.
F2812	**ENDIF missing**
	Not all **IF ENDIF** blocks were exited before an **END** statement appeared.
F2813	**DO-loop label** *number* **: not seen**
	Not all **DO** loops were exited before an **END** statement appeared.
F2814	**IF, ELSEIF, or ELSE missing**
	No **IF**, **ELSEIF**, or **ELSE** statement matching an **ENDIF** statement appeared in the program.
F2815	**assigned GOTO variable not INTEGER**
	Only items of type **INTEGER***n* and **INTEGER** are legal for assigned **GOTO** variables.
F2816	**computed GOTO variable not INTEGER**
	Only items of type **INTEGER***n* and **INTEGER** are legal for computed **GOTO** variables.
F2817	**expression type not LOGICAL**
	Expression types for logical or block **IF** statements must be of type **LOGICAL**[[*n*]].
F2818	**expression type not INTEGER or REAL**
	Expression types for arithmetic **IF** statements must be of type **INTEGER**[[*n*]] or **REAL**[[*n*]].

Number	Compiler Compilation Error Message
F2819	**illegal statement after logical IF**

Only single-line statements can follow logical **IF** statements. All executable statements except **DO**, **ELSE**, **ELSEIF**, **END**, **ENDIF**, block **IF**, and logical **IF** can follow a logical **IF** statement.

F2820 **block label** *number* **: must not be referenced**

Labels that appear on **ELSE** and **ELSEIF** statements cannot be referenced.

F2821 **label** *number* **: previously used as executable label**

The specified label, previously referenced as an executable label, was used as a label for a **FORMAT** statement or specification statement.

F2822 **label** *number* **: previously used as FORMAT label**

The specified label, previously referenced as a label for a **FORMAT** statement, was used as an executable label or a label for a specification statement.

F2823 **DO-loop label** *number* **: out of order**

The specified termination label for a **DO** statement was out of order. **DO**-loop labels may have been reversed.

F2824 **assigned and unconditional GOTO illegal here**

Assigned and unconditional **GOTO** statements cannot terminate **DO** loops.

F2825 **block and arithmetic IF illegal here**

Block and arithmetic **IF** statements cannot terminate a **DO** loop.

F2826 **statement illegal as DO-loop termination**

An **ELSE**, **ELSEIF**, **END**, **ENDIF**, **FORMAT**, **RETURN**, or **STOP** statement cannot be used to terminate a **DO** loop.

F2827 **STOP (or PAUSE) : maximum of 5 digits**

The **STOP** and **PAUSE** statements allow only numeric values between 0 and 99,999, inclusive.

F2828 **ASSIGN target not an INTEGER variable**

Only variables of type **INTEGER** are allowed as targets in **ASSIGN** statements.

F2829 **STOP (or PAUSE) : illegal expression**

Only integers or character constants are legal in **STOP** and **PAUSE** statements.

Number	Compiler Compilation Error Message
F2830	**END missing**
	An **END** statement did not appear as the last statement in the module.
F2831	**label** *number* **: must not be referenced**
	The specified label appeared on a specification or **DATA** statement.
F2832	**statement illegal in INTERFACE**
	Only specification statements are legal in **INTERFACE** statements.
F2833	**RETURN : integer or character expression required**
	If the /4Ys compiler option is used for compiling (or the **$STRICT** metacommand is in effect), only integer or character expressions can follow the **RETURN** statement.
F2834	*name* **: alternate RETURN missing**
	An alternate **RETURN** statement was given in the specified subprogram when none was given in the subprogram declaration.
F2835	**statement out of order or END missing**
	A specification statement was embedded in execution statements, another statement appeared out of the legal statement sequence, or an **END** statement did not terminate a previous subprogram.
F2836	**statement out of order**
	A statement appeared out of the legal order of statements in the program. For example, a specification statement may have appeared with execution statements.
F2837	**label** *number* **: undefined**
	The specified label, which was referenced in a subprogram, was not defined.
F2838	**statement illegal in BLOCK DATA**
	Only type-specification and **DATA** statements are legal in **BLOCK DATA** subprograms.
F2839	**only variables allowed in assigned GOTO statements**
	Only variables are allowed in assigned **GOTO** statements.
F2840	*name* **: assumed-size array : not reference argument**
	Assumed-size arrays must be passed by reference. They cannot be local entities to the subprogram.

Number	Compiler Compilation Error Message
F2841	*name* **: adjustable-size array : not reference argument**

Adjustable-size arrays must be passed by reference. They cannot be local entities to a subprogram.

F2842 **too many assigned GOTO statements**

Only 255 assigned **GOTO** statements are allowed in a subprogram. (Compile-time error.)

F2843 **statement illegal with INTERFACE TO**

An interface must refer to a subroutine or function. The **INTERFACE TO** statement may not be used to declare calls to subprograms. For more information, see the entry for the **INTERFACE TO** statement in Section 4.2, "Statement Directory."

F2844 **no matching DO loop**

An **EXIT**, **CYCLE**, or **END DO** was seen without a matching **DO** loop.

F2845 **END SELECT missing**

A **SELECT** construct was not closed by the end of the subprogram.

F2846 **DO-LOOP ENDDO : not seen**

A **DO** loop without a label was not closed by the end of the subprogram.

F2847 **statement illegal in STRUCTURE declaration**

Only **STRUCTURE** statements are allowed in a **STRUCTURE** declaration.

F2860 **expression must be integer, character, or logical**

SELECT CASE expressions must be integer, **CHARACTER*1**, or logical.

F2861 **no matching SELECT CASE statement**

An **END SELECT** statement was seen without a previous **SELECT CASE** statement.

F2862 **only one CASE DEFAULT allowed**

No more than one **CASE DEFAULT** can appear in a **SELECT CASE** statement.

F2863 **CASE values must be constant expressions**

A **CASE** value cannot be a variable or an expression that contains variables.

Number	Compiler Compilation Error Message
F2864	**CASE value type does not match SELECT CASE expression type** A **CASE** value type and **SELECT CASE** expression data types must match.
F2865	**overlapping case values** A given value must match only one **CASE**.
F2866	**The CASE statement must follow a SELECT CASE statement** A **CASE** statement appeared without a previous **SELECT CASE** statement.
F2867	**LOGICAL case value ranges illegal** A logical case value must not be expressed as a range.
F2868	**SELECT CASE : character expression must be of length 1** Only **CHARACTER*1** character expressions are allowed.
F2869	**lower value exceeds upper value in case value range** The value to the left of the colon must be less than the value to the right of the colon.
F2870	*name* **: element is an array** A field which is an array of structures is not being indexed.
F2901	**-4I2 or -4I4 expected** Only 2- and 4-byte default integer and logical values are supported.
F2902	**-4Y and -4N : both options used for argument** The **$DO66** or **$FREEFORM** metacommand was specified in both the /4Y and the /4N compiler options.
F2993	**separator expected in format** When the /4Ys compiler option is used (or the **$STRICT** metacommand is in effect), a comma (,), colon (:), right parenthesis ( )), or slash (/) is expected to separate items in a format except in the following cases: 1. Between a **P** edit descriptor and an immediately following **F**, **D**, **E**, or **G** edit descriptor 2. Before or after a slash (/) edit descriptor 3. Before or after a colon (:) edit descriptor

Number	Compiler Compilation Error Message
F2994	**\ or $: nonstandard edit descriptor in format**

The \ and $ edit descriptors are not part of standard FORTRAN 77 but are extensions to the language. This error occurs only if the /4Ys compiler option is used (or if the **$STRICT** metacommand is in effect).

F2995 **Z : nonstandard edit descriptor in format**

The Z edit descriptor is not part of standard FORTRAN 77 but is an extension to the language. This error occurs only if the /4Ys compiler option is used (or if the **$STRICT** metacommand is in effect).

F2999 **Hollerith illegal with CHARACTER in relationals**

Hollerith constants cannot be mixed with character variables or constants in relational equations. For example, if `3HYES` is a Hollerith constant and `STR` is a **CHARACTER*N** variable, then the line

```
3HYES .LT. STR              ! Error!
```

is illegal.

Replace the Hollerith with a **CHARACTER** literal (`'YES'` in the example above).

D.2.3 Recoverable Error Messages

The messages listed below indicate potential problems but do not hinder compilation and linking. The /W compiler option has no effect on the output of these messages.

Number	Compiler Recoverable Error Message
F3606	*subprogram* **: formal argument** *name* **: type mismatch**

The type of a formal argument was different from the type of the actual argument used in the subprogram call.

F3607 *subprogram* **: formal argument** *name* **: length mismatch**

The length of a formal argument was different from the length of the actual argument used in the subprogram call.

D.2.4 Warning Error Messages

The messages listed below indicate potential problems but do not hinder compilation and linking.

Number	Compiler Warning Error Message
F4000	**UNKNOWN WARNING** **Contact Microsoft Technical Support** An unknown warning has occurred. Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F4001	**colon expected following ALIAS** An **ALIAS** attribute had the wrong form. The correct form for **ALIAS** is the following: **ALIAS:***string*
F4002	**$DEBUG:'<debug-list>' illegal with $FREEFORM** This form of the **$DEBUG** metacommand was used when the **$FREEFORM** metacommand was in effect.
F4003	**$DECMATH not supported** The **$DECMATH** metacommand is not supported in this version of FORTRAN.
F4006	**metacommand already set** A metacommand that may appear only once was reset.
F4007	**metacommand must come before all FORTRAN statements** This metacommand must appear before all FORTRAN statements.
F4008	**characters following metacommand ignored** Any characters that follow a fully processed metacommand are ignored.
F4010	*filename* **: error closing file** A system error occurred while the specified source file was being closed.
F4011	**empty escape sequence** A backslash (\ ) occurred at the end of a C string such as `'abc\'`. It is replaced by a zero. The backslash should be removed.

Number	Compiler Warning Error Message
F4014	*character* **: nonalphabetic character in $DEBUG ignored**

A nonalphabetic character was included in the list for the **$DEBUG** metacommand.

The characters a–z or A–Z are the only legal characters. Case is ignored.

F4056 **overflow in constant arithmetic**

The result of an operation exceeded #7FFFFFFF.

F4057 **overflow in constant multiplication**

The result of an operation exceeded #7FFFFFFF.

F4058 **address of frame variable taken, DS != SS**

The program was compiled with the default data segment (**DS**) not equal to the stack segment (**SS**), and the program tried to point to a frame variable with a near pointer.

F4059 **segment lost in conversion**

The conversion of a far pointer (a full segmented address) to a near pointer (a segmented offset) resulted in the loss of the segmented address.

F4060 **conversion of long address to short address**

The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) resulted in the loss of the segmented address.

F4061 **long/short mismatch in argument : conversion supplied**

Actual and formal arguments of a function differed in base type. The type of the actual argument was converted to the type of the formal argument.

F4062 **near/far mismatch in argument : conversion supplied**

Actual and formal arguments of a function differed in pointer size. The size of the actual argument was converted to the size of the formal argument.

F4063 *name* **: function too large for post-optimizer**

The compiler tried to optimize a function but ran out of memory while doing so. It flagged the warning, skipped the optimization, and continued the compilation.

To avoid this problem, break the functions in the program into smaller functions.

Number	Compiler Warning Error Message
F4064	**procedure too large, skipping** *optimization* **optimization and continuing**

The compiler tried to perform the given type of optimization on a function but ran out of memory while doing so. It flagged the warning, skipped the given part of the optimization, and continued the compilation.

To avoid this problem, break the function into smaller functions.

F4065 **recoverable heap overflow in post-optimizer - some optimizations may be missed**

The compiler tried to optimize a function but ran out of memory while doing so. It flagged the warning, skipped the optimization, and continued the compilation.

To avoid this problem, break the function into smaller functions.

F4066 **local symbol table overflow - some local symbols may be missing in listings**

The compiler ran out of memory when it tried to collect the local symbols for source listings. Not all of the symbols are listed.

F4072 **insufficient memory to process debugging information**

You specified the /Zi compiler option, but the compiler did not have enough memory to store all of the required debugging information. (Compile-time warning.)

F4186 **string too long – truncated to 40 characters**

A string of more than 40 characters was used in a **$TITLE** or **$SUBTITLE** meta-command. The string is truncated to 40 characters.

F4201 **ENTRY : formal argument** *name* **: ATTRIBUTE** *attribute* **: mismatch**

VALUE and **REFERENCE** attributes were mismatched in the declaration and use of an **ENTRY** statement.

F4202 *subprogram* **: formal argument** *name* **: never used**

If a formal argument is never referenced, the compiler must assume a variable was meant for this argument. In medium model, if a function is passed to the formal argument, the wrong amount of storage may be allocated. This message is suppressed by any previous compiler error message (F2*XXX*).

F4303 *name* **: language attributes illegal on formal arguments**

A language attribute (**C** or **PASCAL**) was specified for a formal argument to the current routine. It has no effect.

Number	Compiler Warning Error Message
F4313	*name* : **not previously declared** While the /4Yd compiler option was used (or the **$DECLARE** metacommand was in effect), *name* was not declared in a type statement before it was used.
F4314	*intrinsic* : **declared with wrong type** The specified name was declared with an incorrect type in an **INTRINSIC** statement. The incorrect type is ignored, and the correct type is used.
F4315	*name* : *attribute* **illegal with attributes specified in same list** The specified attribute contradicts an earlier attribute for the item in the same attribute list.
F4316	*name* : *attribute* **illegal with attributes specified in earlier list** The specified attribute contradicts an attribute in an earlier attribute list for the item.
F4317	*name* : *attribute* **attribute repeated** The specified attribute was already used once in an earlier attribute list for the item, and it should only have appeared in one attribute list.
F4318	*name* : *attribute* **illegal on COMMON statements** The specified attribute is illegal on common-block declarations.
F4319	*name* : *attribute* **illegal on formal arguments** The specified attribute cannot be used on formal arguments.
F4320	*name* : *attribute* **illegal on ENTRY statements** The specified attribute cannot be used on **ENTRY** statements.
F4321	*name* : *attribute* **illegal on subprogram statements** The specified attribute cannot be used on **SUBPROGRAM** statements.
F4322	*name* : *attribute* **illegal on variable declarations** The specified attribute cannot be used on variable declarations.
F4323	*name* : *attribute* **illegal on type declarations** The specified attribute cannot be used on type declarations.
F4325	*name* : *attribute* **illegal on NAMELIST declarations** Attributes are illegal on **NAMELIST** declarations.

Number Compiler Warning Error Message

F4326 *name* **: EQUIVALENCE : nonconstant upper substring expression ignored**

The upper substring expression in an **EQUIVALENCE** statement was not a constant. Since the expression is not used in the addressing expression, it is ignored.

F4327 *name* **: INTERFACE : not formal argument**

A variable was declared that was not given in the formal-argument list to the subprogram specified in the **INTERFACE** statement.

F4328 *name* **:** *attribute* **illegal on STRUCTURE declarations**

An illegal attribute was specified on a **STRUCTURE** declaration.

F4329 **%fs : COMMON : size changed**

This is a warning level message which is the same as F2323. F2323 now only occurs if **$STRICT** (or -4Ys) is set. Otherwise, the warning occurs.

F4330 *varname* **: NEAR/FAR/HUGE equivalence attribute conflict**

A variable or array was declared with one attribute but equivalenced into a block with a different attribute. For instance,

```
INTEGER A(100000)
INTEGER B[NEAR](5)
EQUIVALENCE (A(1), B(1))       ! Error
END
```

A has an implicit attribute of **HUGE** because of its size. B is explicitly declared as **NEAR**, however, causing an attribute conflict when the line

```
EQUIVALENCE (A(1), B(1))       ! Error
```

tries to make A and B equivalent.

F4400 **DATA : more constants than names**

Extra constants appearing in a constant list of a **DATA** statement were ignored.

F4501 *array* **: subscript** *number* **out of range**

The /4Yb compiler option was used in compiling (or the **$DEBUG** metacommand was in effect), and an array passed as an argument had a bound out of range. (This practice is legal for formal arguments because it is common in FORTRAN to declare the last bound to be 1.)

F4602 *name* **: alternate RETURN statement missing**

The subprogram declaration where the specified name appeared had no alternate **RETURN** statement.

Number	Compiler Warning Error Message
F4605	*name* **: FAR formal argument** *name* **: passed HUGE array**

An array declared with a **HUGE** attribute was passed to a formal argument declared with a **FAR** attribute.

F4608 *name* **: formal argument** *name* **: passed FAR/HUGE**

A variable which was declared **NEAR** is being passed **FAR**.

F4801 **label** *number* **: used across blocks**

An executable statement label was referenced across a statement block. This situation may arise in the following cases:

- When a **GOTO** statement uses a statement label in a different arm of an **IF...ELSE...ENDIF** statement
- When the program jumps into a **DO** loop

F4802 **no assigned GOTO or FMT= for ASSIGN statement**

An **ASSIGN** statement was used to assign a label to a variable in the subprogram, but the variable was not used.

F4803 *name* **: FUNCTION : return variable not set**

A return variable specified in a **FUNCTION** statement was not set at least once in the function.

F4901 **-4Y and -4N : both options used; -4Y assumed**

The **$DEBUG**, **$DECLARE**, **$LIST**, **$STRICT**, or **$TRUNCATE** metacommand was specified with both the /4Y and /4N compiler options. For example, **$DEBUG** was specified using both a /4Yb option and a /4Nb option.

F4902 **-W***number* **: illegal warning level ignored**

This is an internal check. Microsoft FORTRAN supports only warning levels 0 and 1.

F4903 **-Zp***number* **: illegal pack value ignored**

Only the structure packing values 1, 2, and 4 are valid.

F4980 **integer expected in format**

An edit descriptor lacked a required integer value.

F4981 **initial left parenthesis expected in format**

A format did not start with a left parenthesis (( ).

Number	Compiler Warning Error Message

F4982 **positive integer expected in format**

An unexpected negative or 0 value was used in a format.

Negative integer values can appear only with the **P** edit descriptor. Integer values of 0 can appear only in the *d* and *m* fields of numeric edit descriptors.

F4983 **repeat count on nonrepeatable descriptor**

One or more **BN**, **BZ**, **S**, **SP**, **SS**, **T**, **TL**, **TR**, /, \, **$**, :, or apostrophe (') edit descriptors had repeat counts associated with them.

F4984 **integer expected preceding H, X, or P edit descriptor**

An integer did not precede a (nonrepeatable) **H**, **X**, or **P** edit descriptor.

The correct formats for these edit descriptors are *n***H**, *n***X**, and *k***P**, respectively, where *n* is a positive integer and *k* is an optionally signed integer.

F4985 **N or Z expected after B in format**

An illegal edit descriptor beginning with "B" was used.

The only valid edit descriptors beginning with "B" are **BN** and **BZ**, used to specify the interpretation of blanks as nulls or zeros, respectively.

F4986 **format nesting limit exceeded**

More than 16 sets of parentheses were nested inside the main level of parentheses in a format.

F4987 **'.' : expected in format**

A period did not appear between the *w* and *d* fields of a **D**, **E**, **F**, or **G** edit descriptor.

F4988 **unexpected end of format**

An incomplete format was used.

Improperly matched parentheses, an unfinished Hollerith (**H**) descriptor, or another incomplete descriptor specification can cause this error.

F4989 **'*character*' : unexpected character in format**

A character that cannot be interpreted as a valid edit descriptor was used in a format.

F4990 **M field exceeds W field in I edit descriptor**

The length of the *m* field specified in an **I** edit descriptor exceeded the length of the *w* field.

Number	Compiler Warning Error Message
F4997	**CHARACTER*(*) in multithread-thread may not work**

Character*(*) arguments and functions may not work when linked with the multithread-thread FORTRAN run-time library.

F4998 *name* **: variable used but not defined**

A variable was used in the subprogram but never given a value.

F4999 *name* **: variable declared but not used**

A variable was declared, but never referenced anywhere else in the subprogram. (This is one of only two warnings shut off by **-W2**.)

D.3 Run-Time Error Messages

Run-time error messages fall into two categories:

1. Error messages generated by the run-time library to notify you of serious errors. These messages are listed and described in Section D.3.1.
2. Floating-point exceptions generated by the 8087/80287 hardware or the emulator. These exceptions are listed and described in Section D.3.2, "Other Run-Time Error Messages."

D.3.1 Run-Time-Library Error Messages

The following messages may appear at run time when your program has serious errors. Run-time error-message numbers range from F6000 to F6999.

A run-time error message takes the following general form:

[[*sourcefile*(*line*) :]] run-time error F6*nnn* : *operation*[| (*filename*)]]
- *messagetext*

The *sourcefile* (*line*) information appears only when the **$DEBUG** metacommand is in effect.

For *operation*, one of the following may appear: **ALLOCATE**, **BACKSPACE**, **BEGINTHREAD**, **CLOSE**, **DEALLOCATE**, **ENDFILE**, **EOF**, **INQUIRE**, **LOCKING**, **OPEN**, **READ**, **REWIND**, **WRITE**, or **$DEBUG**.

The *filename* of the file affected by *operation* is shown except when *operation* is **$DEBUG**.

The *messagetext* follows on the next line.

Number	Run-Time Error Message

F6096 **array subscript expression out of range**

An expression used to index an array was smaller than the lower dimension bound or larger than the upper dimension bound. This message appears only if the /4Yb option is used in compiling (or the **$DEBUG** metacommand is in effect).

F6097 **CHARACTER substring expression out of range**

An expression used to index a character substring was illegal. This message appears only if the /4Yb option is used in compiling (or the **$DEBUG** metacommand is in effect).

F6098 **label not found in assigned GOTO list**

The label assigned to the integer-variable name was not specified in the label list of the assigned **GOTO** statement. This message appears only if the /4Yb option is used in compiling (or the **$DEBUG** metacommand is in effect).

F6099 **INTEGER overflow**

This error occurs whenever integer arithmetic results in overflow, or when assignment to an integer is out of range. This message appears only if the /4Yb option is used in compiling (or the **$DEBUG** metacommand is in effect).

F6100 **INTEGER overflow on input**

An **INTEGER*n** item exceeded the legal size limits.

An **INTEGER*1** item must be in the range –127 to 127. An **INTEGER*2** item must be in the range –32,767 to 32,767. An **INTEGER*4** item must be in the range –2,147,483,647 to 2,147,483,647.

F6101 **invalid INTEGER**

Either an illegal character appeared as part of an integer, or a numeric character larger than the radix was used in an alternate radix specifier.

F6102 **REAL indefinite (uninitialized or previous error)**

An illegal argument was specified for an intrinsic function (for example, `SQRT(-1)` or `ASIN(2)`). This error message does not always appear where the mistake was originally made. It may appear if the invalid value is used later in the program.

F6103 **invalid REAL**

An illegal character appeared as part of a real number.

Number	Run-Time Error Message

F6104 **REAL math overflow**

A real value was too large. Floating-point overflows in either direct or emulated mode generate NAN (Not-A-Number) exceptions, which appear in the output field as asterisks (*) or the letters NAN.

F6200 **formatted I/O not consistent with OPEN options**

The program tried to perform formatted I/O on a unit opened with **FORM='UNFORMATTED'** or **FORM='BINARY'**.

F6201 **list-directed I/O not consistent with OPEN options**

The program tried to perform list-directed I/O on a file that was not opened with **FORM='FORMATTED'** and **ACCESS='SEQUENTIAL'**.

F6202 **terminal I/O not consistent with OPEN options**

The **ACCESS='SEQUENTIAL'** option and either the **FORM='FORMATTED'** or the **FORM='BINARY'** option were not included in the **OPEN** statement for a special device name such as **CON**, **LPT1**, or **PRN**. These options are required because special device names are connected to devices that do not support direct access.

When a unit is connected to a terminal device, an **OPEN** statement that has the options **FORM='FORMATTED'** and **ACCESS='SEQUENTIAL'** results in carriage control. If the **FORM='BINARY'** and **ACCESS='SEQUENTIAL'** options appear in an **OPEN** statement, binary data transfer takes place.

F6203 **direct I/O not consistent with OPEN options**

A **REC=** option was included in a statement that transferred data to a file that was opened with the **ACCESS='SEQUENTIAL'** option.

F6204 **unformatted I/O not consistent with OPEN options**

If a file is opened with **FORM='FORMATTED'**, unformatted or binary data transfer is prohibited.

F6205 **A edit descriptor expected for CHARACTER**

The **A** edit descriptor was not specified when a character data item was read or written using formatted I/O.

F6206 **E, F, D, or G edit descriptor expected for REAL**

The **E**, **F**, **D**, or **G** edit descriptor was not specified when a real data item was read or written using formatted I/O.

Number	Run-Time Error Message

F6207 **I edit descriptor expected for INTEGER**

The **I** edit descriptor was not specified when an integer data item was read or written using formatted I/O.

F6208 **L edit descriptor expected for LOGICAL**

The **L** edit descriptor was not specified when a logical data item was read or written using formatted I/O.

F6209 **file already open : parameter mismatch**

An **OPEN** statement specified a connection between a unit and a file name that was already in effect. In this case, only the **BLANK=** option can have a different setting.

F6210 **namelist I/O not consistent with OPEN options**

The program tried to perform namelist I/O on a file that was not opened with **FORM='FORMATTED'** and **ACCESS='SEQUENTIAL'**.

F6211 **IOFOCUS illegal with non-window unit**

The **IOFOCUS** parameter can only be used with a unit opened as a window (opened with **FILE = 'USER'** and linked with the QuickWin library). Either open the specified unit as a window, or remove the **IOFOCUS** parameter from the **OPEN** statement.

F6212 **IOFOCUS illegal without QuickWin**

Specifying the **IOFOCUS** parameter requires that you link the program with the QuickWin library.

Relink the program with the QuickWin library.

F6213 **TITLE illegal with non-window unit.**

A title can only be specified for units opened as windows (opened with **FILE = 'USER'** and linked with the QuickWin library). Either reopen the specified unit as a window, or remove the **TITLE** parameter from the **OPEN** statement.

F6214 **TITLE illegal without QuickWin**

Specifying the **TITLE** parameter requires that you link the program with the QuickWin library.

Relink the program with the QuickWin library.

Number	Run-Time Error Message

F6300 **KEEP illegal for scratch file**

STATUS='KEEP' was specified for a scratch file; this is illegal because scratch files are automatically deleted at program termination.

F6301 **SCRATCH illegal for named file**

STATUS='SCRATCH' should not be used in an **OPEN** statement that includes a file name.

F6302 **multiple radix specifiers**

More than one alternate radix for numeric I/O was specified.

F6303 **illegal radix specifier**

A radix specifier was not between 2 and 36, inclusive.

F6304 **illegal STATUS value**

An illegal value was used with the **STATUS=** option.

STATUS= accepts the following values:

1. **'KEEP'** or **'DELETE'** when used with **CLOSE** statements
2. **'OLD'**, **'NEW'**, **'SCRATCH'**, or **'UNKNOWN'** when used with **OPEN** statements

F6305 **illegal MODE value**

An illegal value was used with the **MODE=** option.

MODE= accepts the values **'READ'**, **'WRITE'**, or **'READWRITE'**.

F6306 **illegal ACCESS value**

An illegal value was used with the **ACCESS=** option.

ACCESS= accepts the values **'SEQUENTIAL'** and **'DIRECT'**.

F6307 **illegal BLANK value**

An illegal value was used with the **BLANK=** option.

BLANK= accepts the values **'NULL'** and **'ZERO'**.

Number	Run-Time Error Message
F6308	**illegal FORM value**

An illegal value was used with the **FORM=** option.

FORM= accepts the following values: **'FORMATTED'**, **'UNFORMATTED'**, and **'BINARY'**.

F6309 **illegal SHARE value**

An illegal value was used with the **SHARE=** option.

SHARE= accepts the values **'COMPAT'**, **'DENYRW'**, **'DENYWR'**, **'DENYRD'**, and **'DENYNONE'**.

F6310 **illegal LOCKMODE value**

An illegal value was used with the **LOCKMODE=** option.

LOCKMODE= accepts the values **'LOCK'**, **'NBLCK'**, **'NBRLCK'**, **'RLCK'**, and **'UNLCK'**.

F6311 **illegal record number**

An invalid number was specified as the record number for a direct-access file.

The first valid record number for direct-access files is 1.

F6312 **no unit number associated with ***

In an **INQUIRE** statement, the **NUMBER=** option was specified for the file associated with * (console).

F6313 **illegal RECORDS value**

The **RECORDS=** option in a **LOCKING** statement specified a negative number.

F6314 **illegal unit number**

An illegal unit number was specified.

Legal unit numbers can range from –32,767 to 32,767, inclusive.

F6315 **illegal RECL value**

A negative or zero record length was specified for a direct file.

The smallest valid record length for direct files is 1.

F6316 **array already allocated**

The program attempted to reallocate an already allocated array.

Number	Run-Time Error Message
F6317	**array size zero or negative**

The size specified for an array in an **ALLOCATE** statement must be greater than zero.

F6318 **non-HUGE array exceeds 64K**

The memory space required for an array in an **ALLOCATE** statement exceeds 64K, but the **HUGE** attribute was not specified.

F6319 **array not allocated**

The program attempted to **DEALLOCATE** an array that was never allocated.

F6400 **BACKSPACE illegal on terminal device**

A **BACKSPACE** statement specified a unit connected to a terminal device such as a terminal or printer.

F6401 **EOF illegal on terminal device**

An **EOF** intrinsic function specified a unit connected to a terminal device such as a terminal or printer.

F6402 **ENDFILE illegal on terminal device**

An **ENDFILE** statement specified a unit connected to a terminal device such as a terminal or printer.

F6403 **REWIND illegal on terminal device**

A **REWIND** statement specified a unit connected to a terminal device such as a terminal or printer.

F6404 **DELETE illegal for read-only file**

A **CLOSE** statement specified **STATUS='DELETE'** for a read-only file.

F6405 **external I/O illegal beyond end of file**

The program tried to access a file after executing an **ENDFILE** statement or after it encountered the end-of-file record during a read operation.

A **BACKSPACE**, **REWIND**, or **OPEN** statement must be used to reposition the file before execution of any I/O statement that transfers data.

Number	Run-Time Error Message
F6406	**truncation error : file closed**

This is a transient error. While the file was being truncated, it was temporarily closed.

After a few minutes, the file should be run again. If this error message reappears, the file should be checked for characteristics, such as locking or permissions, that would prevent it from being accessed.

F6407 **terminal buffer overflow**

More than 131 characters were input to a record of a unit connected to the terminal (keyboard). Note that the operating system may impose additional limits on the number of characters that can be input to the terminal in a single record.

F6408 **comma delimiter disabled after left repositioning**

A comma could not be used as a field delimiter. This is because the use of commas as input field delimiters is disabled if left tabbing leaves the file positioned in a previous buffer.

F6409 **LOCKING illegal on sequential file**

A **LOCKING** statement specified a unit that was not opened with **ACCESS='DIRECT'**.

F6410 **file already locked or unlocked**

The program tried to lock a file that was already locked or tried to unlock a file that was already unlocked.

F6411 **file deadlocked**

A **LOCKING** statement that included the **'LOCK'** or **'RLCK'** value tried to lock a file, but the file could not be locked after 10 attempts.

F6412 **SHARE not installed**

The SHARE.COM or SHARE.EXE file must be installed on your system before you can use the **LOCKING** statement, or the **SHARE=** option in an **OPEN** statement.

F6413 **file already connected to a different unit**

The program tried to connect an already connected file to a new unit.

A file can be connected to only one unit at a time.

Number	Run-Time Error Message

F6414 **access not allowed**

This error is caused by one of the following occurrences:

- The file name specified in an **OPEN** statement was a directory.
- An **OPEN** statement tried to open a read-only file for writing.
- The file's sharing mode does not allow the specified operations (DOS Versions 3.0 and later only).

F6415 **file already exists**

An **OPEN** statement specified **STATUS='NEW'** for a file that already exists.

F6416 **file not found**

An **OPEN** statement specified **STATUS='OLD'** for a file that does not exist.

F6417 **too many open files**

The program exceeded the system limit on the number of open files allowed at one time.

To fix this problem, change the **FILES=** command in the CONFIG.SYS file.

F6418 **too many units connected**

The program exceeded the limit on the number of open files per program.

Close any unnecessary files. See the **FILES=** command in the *Microsoft MS-DOS User's Guide and User's Reference* for more information.

F6419 **illegal structure for unformatted file**

The file was opened with **FORM='UNFORMATTED'** and **ACCESS='SEQUENTIAL'**, but its internal physical-record structure was incorrect or inconsistent.

F6420 **unknown unit number**

A statement such as **BACKSPACE** or **ENDFILE** specified a file that had not yet been opened. (The **READ** and **WRITE** statements do not cause this problem since, instead of generating this error, they prompt you for a file if the file has not been opened yet.)

F6421 **file read-only or locked against writing**

The program tried to transfer data to a file that was opened in read-only mode or locked against writing.

Number	Run-Time Error Message

F6422 **no space left on device**

The program tried to transfer data to a file residing on a device that was out of storage space.

F6423 **too many threads**

The program attempted to execute more threads than the FORTRAN run-time system can handle.

F6424 **invalid argument**

The system could not begin a thread of execution because an argument to the BEGINTHREAD routine is incorrect. This usually occurs when the stack argument does not start on a word boundary (even address), or the stack size argument is odd or zero, or the stack crosses a segment boundary.

F6425 **BACKSPACE illegal for SEQUENTIAL WRITE-ONLY files**

The **BACKSPACE** statement is not allowed in files opened with **STATUS = WRITE** (write-only status) since **BACKSPACE** requires reading the previous record in the file to provide positioning.

Resolve the problem by giving the file read access or by avoiding the **BACKSPACE** statement. Note that the **REWIND** statement is valid for files opened as write-only.

F6500 **file not open for reading or file locked**

The program tried to read from a file that was not opened for reading or was locked.

F6501 **end of file encountered**

The program tried to read more data than the file contains.

F6502 **positive integer expected in repeat field**

When the *r***c* form is used in list-directed input, the *r* must be a positive integer.

F6503 **multiple repeat field**

In list-directed input of the form *r**c, an extra repeat field was used. For example,

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

with input `2*1*3` returns this error. The `2*1` means send two values, each 1; the `*3` is an error.

Number	Run-Time Error Message

F6504 **invalid number in input**

Some of the values in a list-directed input record were not numeric. The following example would cause this error:

```
123abc
```

F6505 **invalid string in input**

A string item was not enclosed in single quotation marks.

F6506 **comma missing in COMPLEX input**

When using list-directed input, the real and imaginary components of a complex number were not separated by a comma.

F6507 **T or F expected in LOGICAL read**

The wrong format was used for the input field for logical data.

The input field for logical data consists of optional blanks, followed by an optional decimal point, followed by a **T** for true or **F** for false. The **T** or **F** may be followed by additional characters in the field, so that **.TRUE.** and **.FALSE.** are acceptable input forms.

F6508 **too many bytes read from unformatted record**

The program tried to read more data from an unformatted file than the current record contained. If the program was reading from an unformatted direct file, it tried to read more than the fixed record length as specified by the **RECL=** option. If the program was reading from an unformatted sequential file, it tried to read more data than was written to the record.

F6509 **H or apostrophe edit descriptor illegal on input**

Hollerith (**H**) or apostrophe edit descriptors were encountered in a format used by a **READ** statement.

F6510 **illegal character in hexadecimal input**

The input field contained a character that was not hexadecimal.

Legal hexadecimal characters are 0–9 and A–F.

F6511 **variable name not found**

A name encountered on input from a namelist record is not declared in the corresponding **NAMELIST** statement.

F6512 **invalid NAMELIST input format**

The input record is not in the correct form for namelist input.

Number	Run-Time Error Message
F6513	**wrong number of array dimensions**

In namelist input, an array name was qualified with a different number of subscripts than its declaration, or a non-array name was qualified.

F6514 **array subscript exceeds allocated area**

A subscript was specified in namelist input which exceeded the declared dimensions of the array.

F6515 **invalid subrange in namelist input**

A character item in namelist input was qualified with a subrange that did not meet the requirement that 1 <= e1 <= e2 <= len (where 'len' is the length of the character item, 'e1' is the leftmost position of the substring, and 'e2' is the rightmost position of the substring).

F6516 **substring range specified on non-CHARACTER item**

A non-character item in namelist input was qualified with a substring range.

F6600 **internal file overflow**

The program either overflowed an internal-file record or tried to write to a record beyond the end of an internal file.

F6601 **direct record overflow**

The program tried to write more than the number of bytes specified in the **RECL=** option to an individual record of a direct-access file.

F6602 **numeric field bigger than record size**

The program tried to write a noncharacter item across a record boundary in list-directed or namelist output. Only character constants can cross record boundaries.

F6700 **heap space limit exceeded**

The program tried to open too many files at once. A file control block (FCB) must be allocated from the heap for each file opened, but no more heap space was available.

F6701 **scratch file name limit exceeded**

The program exhausted the template used to generate unique scratch-file names.

F6980 **integer expected in format**

An edit descriptor lacked a required integer value.

Number	Run-Time Error Message
F6981	**initial left parenthesis expected in format**

A format did not begin with a left parenthesis (( ).

F6982 **positive integer expected in format**

A zero or negative integer value was used in a format.

Negative integer values can appear only with the **P** edit descriptor. Integer values of 0 can appear only in the *d* and *m* fields of numeric edit descriptors.

F6983 **repeat count on nonrepeatable descriptor**

One or more **BN**, **BZ**, **S**, **SS**, **SP**, **T**, **TL**, **TR**, **/**, ****, **$**, **:**, or apostrophe (**'**) edit descriptors had repeat counts associated with them.

F6984 **integer expected preceding H, X, or P edit descriptor**

An integer did not precede a (nonrepeatable) **H**, **X**, or **P** edit descriptor.

The correct formats for these descriptors are *n***H**, *n***X**, and *k***P**, respectively, where *n* is a positive integer and *k* is an optionally signed integer.

F6985 **N or Z expected after B in format**

An illegal edit descriptor beginning with "B" was used.

The only valid edit descriptors beginning with "B" are **BN** and **BZ**, used to specify the interpretation of blanks as nulls or zeros, respectively.

F6986 **format nesting limit exceeded**

More than 16 sets of parentheses were nested inside the main level of parentheses in a format.

F6987 **'.' expected in format**

No period appeared between the *w* and *d* fields of a **D**, **E**, **F**, or **G** edit descriptor.

F6988 **unexpected end of format**

An incomplete format was used.

Improperly matched parentheses, an unfinished Hollerith (**H**) descriptor, or another incomplete descriptor specification can cause this error.

Number	Run-Time Error Message
F6989	**unexpected character in format**

A character that cannot be interpreted as part of a valid edit descriptor was used in a format.

F6990 **M field exceeds W field in I edit descriptor**

The value of the *m* field specified in an **I** edit descriptor exceeded the value of the *w* field.

F6991 **integer out of range in format**

An integer value specified in an edit descriptor was too large to represent as a 4-byte integer.

F6992 **format not set by ASSIGN**

The format specifier in a **READ**, **WRITE**, or **PRINT** statement was an integer variable, but an **ASSIGN** statement did not properly assign it the statement label of a **FORMAT** statement in the same program unit.

D.3.2 Other Run-Time Error Messages

The following sections describe math run-time errors and general run-time errors. Math run-time errors are divided into low-level and function-level math errors.

Low-Level Math Errors

The error messages listed below correspond to exceptions generated by the 8087/80287 hardware. Refer to the Intel documentation for your processor for a detailed discussion of hardware exceptions. These errors may also be detected by the floating-point emulator or alternate math library.

Using FORTRAN's default 8087/80287 control-word settings, the following exceptions are masked and do not occur:

Exception	Default Masked Action
Denormal	Exception masked
Underflow	Result goes to 0.0
Inexact	Exception masked

See Chapter 1, "Controlling Floating-Point Operations," *in Microsoft FORTRAN Advanced Topics* for information on how to change the floating-point control word.

The following errors do not occur with code generated by the Microsoft FORTRAN Compiler or code provided in the standard Microsoft FORTRAN libraries:

```
square root
stack underflow
unemulated
```

The low-level math error messages, listed below, have the following format:

[[*sourcefile*(*line*) :]] run-time error M61*xx*: MATH
- floating-point error: *messagetext*

The *sourcefile* and *line* where the error occurred appear only if the /4Yb option is used in compiling (or the **$DEBUG** metacommand is in effect).

Number	Low-Level Math Error Message
M6101	**invalid**

An invalid operation occurred. This error usually occurs when operating on a NAN (not a number) or infinity.

This error terminates the program with exit code 129.

M6102 **denormal**

A very small floating-point number was generated which may no longer be valid due to loss of significance. Denormals are normally masked, causing them to be trapped and operated upon.

This error terminates the program with exit code 130.

M6103 **divide by 0**

A floating-point operation attempted to divide by zero.

This error terminates the program with exit code 131.

M6104 **overflow**

An overflow occurred in a floating-point operation.

This error terminates the program with exit code 132.

M6105 **underflow**

An underflow occurred in a floating-point operation. An underflow is normally masked, with the underflowing value replaced by 0.0.

This error terminates the program with exit code 133.

Number Low-Level Math Error Message

M6106 inexact

Loss of precision occurred in a floating-point operation. This exception is normally masked because almost any floating-point operation can cause a loss of precision.

This error terminates the program with exit code 134.

M6107 unemulated

An attempt was made to execute a coprocessor instruction that is invalid or is not supported by the emulator.

This error terminates the program with exit code 135.

M6108 square root

The operand in a square-root operation was negative.

This error terminates the program with exit code 136.

NOTE: The **Sqrt** function in the C run-time library and the FORTRAN intrinsic function **SQRT** do not generate this error. **Sqrt** checks the argument before performing the operation and returns an error value if the operand is negative. **SQRT** generates the DOMAIN error M6201 instead of this error.

M6110 stack overflow

A floating-point expression caused a stack overflow on the 8087/287/387 coprocessor or the emulator. Stack-overflow exceptions are trapped up to a limit of seven levels in addition to the eight levels normally supported by the 8087/287/387 coprocessor.

This error terminates the program with exit code 138.

M6111 stack underflow

A floating-point operation resulted in a stack underflow on the 8087/287/387 coprocessor or the emulator.

This error is often caused by a call to a long double function that does not return a value. For example, the following gives this error when compiled and run:

```
long double ld() {};

main () { ld(); }
```

This error terminates the program with exit code 139.

Number	Low-Level Math Error Message

M6201 *function* **: DOMAIN error**

An argument to the given function was outside the domain of legal input values for that function.

For example, the following expression causes this error:

```
SQRT(-1.0)
```

This error calls the **matherr( )** function with the function name, its arguments, and the error type. You can rewrite **matherr( )**.

M6202 *function* **: SING error**

An argument to the given function was a singularity value for this function. The function is not defined for that argument.

For example, the following expression causes this error:

```
LOG10(0.0)
```

This error calls the **matherr( )** function with the function name, its arguments, and the error type. You can rewrite **matherr( )**.

M6203 *function* **: OVERFLOW error**

The given function result was too large to be represented.

This error calls the **matherr( )** function with the function name, its arguments, and the error type. You can rewrite **matherr( )**.

M6205 *function* **: TLOSS error**

A total loss of significance (precision) occurred.

D.3.3 General Run-Time Error Messages

The following messages indicate general problems that may occur during program start-up, termination, or execution. These error messages have the following format:

[[*sourcefile*(*line*):]] run-time error R6*xxx*
- *messagetext*

The *sourcefile* and the *line* where the error occurred appear only if the /4Yb compiler option is used to compile the program (or the **$DEBUG** metacommand is in effect). This additional information is not available for R6002, R6004, R6008, and R6009, which appear at start-up time.

Number	General Run-Time Error Message

R6000 **stack overflow**

The program has run out of stack space. This can occur when a program uses a large amount of local data or is heavily recursive.

There are several ways to allocate a larger stack:

- Recompile using the /F compiler option.
- Relink using LINK's /STACK option.
- Run EXEHDR on the program using the /STACK option.

R6001 **null pointer assignment**

The contents of the NULL segment have changed in the course of program execution. The program has written to this area, usually by an inadvertent assignment through a null pointer.

The NULL segment is a location in low memory that is not normally used. The contents of the NULL segment are checked upon program termination. If a change is detected, this error is generated.

Note that the program can contain null pointers without causing this error. The error appears only when the program writes to memory through a null pointer. It reflects a potentially serious error in the program. Although a program that produces this error may appear to operate correctly, it may cause problems in the future and may fail to run in a different operating environment.

Number	General Run-Time Error Message

R6002 **floating-point support not loaded**

The program needs the floating-point library, but the library was not linked to the program.

One of the following may have occurred:

- The program was compiled or linked with an option (such as /FPi87) that required a coprocessor, but the program was run on a machine that did not have a coprocessor installed.
- A format string for a printf or scanf function contained a floating-point format specification, and the program did not contain any floating-point values or variables.

  The compiler minimizes a program's size by loading floating-point support only when necessary. It cannot detect floating-point format specifications in format strings, so it does not load the necessary floating-point routines.

  Use a floating-point argument to correspond to the floating- point format specification, or perform a floating-point assignment elsewhere in the program. This causes floating-point support to be loaded.
- In a mixed-language program, a C library was specified before a FORTRAN library when the program was linked. Relink and specify the C library last.

R6003 **integer divide by 0**

An attempt was made to divide an integer by 0, giving an undefined result.

R6005 **not enough memory on exec**

Not enough memory was available to load the program being spawned.

This error occurs when a child process spawned by one of the **exec** library routines fails and the operating system cannot return control to the parent process.

R6006 **invalid format on exec**

The file to be executed by one of the **exec** functions was not in the correct format for an executable file.

This error occurs when a child process spawned by one of the **exec** library routines fails and the operating system cannot return control to the parent process.

Number	General Run-Time Error Message

R6007 **invalid environment on exec**

During a call to an **exec** function, the operating system found that the child process was given an invalid environment block.

This error occurs when a child process spawned by one of the **exec** library routines fails and the operating system cannot return control to the parent process.

R6008 **not enough space for arguments**

There was enough memory to load the program but not enough room for the argv array.

One of the following may be a solution:

- Increase the amount of memory available to the program.
- Reduce the number and size of command-line arguments.
- Reduce the environment size, removing unnecessary variables.
- Rewrite either the **_setargv** or the **_setenvp** routine.

R6009 **not enough space for environment**

There was enough memory to load the program but not enough room for the envp array.

One of the following may be a solution:

- Increase the amount of memory available to the program.
- Reduce the number and size of command-line arguments.
- Reduce the environment size, removing unnecessary variables.
- Rewrite either the **_setargv** or the **_setenvp** routine.

R6010 **abnormal program termination**

This error is displayed by the **abort( )** routine. The program terminates with exit code 3, unless an **abort( )** signal handler has been defined by using the **signal( )** function.

R6012 **illegal near-pointer use**

A null near pointer was used in the program.

This error only occurs if pointer checking is in effect. You can enable pointer checking with either the /Zr compiler option or the check_pointer pragma.

Number	General Run-Time Error Message

R6013 **illegal far-pointer use**

An out-of-range far pointer was used in the program.

This error only occurs if pointer checking is in effect. You can enable pointer checking with either the /Zr compiler option or the check_pointer pragma.

R6016 **not enough space for thread data**

The program could not get enough memory from the operating system to complete a **_beginthread( )** call.

When a new thread is started, the library must create an internal database for that thread. If that database cannot be expanded with memory provided by the operating system, the process cannot continue execution.

R6017 **unexpected multithread lock error**

The process received an unexpected error while trying to access a C run-time multithread lock.

This error usually occurs if the process inadvertently alters the run-time heap data. However, it can also be caused by an internal error in the run-time or operating-system code.

R6018 **unexpected heap error**

The process encountered an unexpected error while performing a memory-management operation.

This error usually occurs if the process inadvertently alters the run-time heap data. However, it can also be caused by an internal error in the run-time or operating-system code.

If your compiler provides a library containing **_heapchk()** and **_heapwalk()**, you can use these functions to diagnose this error.

R6020 **unexpected RTWIN error**

The process encountered an unexpected error pertaining to RTWIN functionality.

One of the following may have occurred:

- The program was built without RTWIN support but tried to access RTWIN functionality.
- An RTWIN operation failed in an unrecoverable manner.

R6021 **no main procedure**

The program does not have a main( ) procedure entry.

Glossary

The definitions in this glossary are intended primarily for use with this manual and *Microsoft FORTRAN Advanced Topics*.

8087, 80287, and 80387 coprocessors Intel® hardware products that provide very fast and precise number processing.

Active page The area of memory that graphics instructions currently write to. This may or may not be the visual page.

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 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 or program 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*; an array has as many dimensions as it has dimension declarators. The number *upper* – *lower* + 1 is the size of a dimension.

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 eight bytes of memory.

Executable program A file containing executable program code. Such files usually end with the .EXE extension. When the name of the file is entered at the system prompt, the instructions in the program are performed.

External In FORTRAN, user-defined subroutines and functions are said to be external (as opposed to the intrinsic procedures that are part of the language). The compiler assumes all references to non-intrinsic procedures are external, and will be satisfied during linkage.

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.

Implicit open The file opening performed by a read or write operation when the file was not explicitly opened by an **OPEN** statement.

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.

Intrinsic A subroutine or function that is part of the FORTRAN language. The compiler knows which procedures are intrinsic, and assumes their references will be linked with code from the FORTRAN libraries.

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 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.

Map An area of memory in which one or more variables are contiguous.

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 or to increase the execution speed by using more efficient algorithms.

Pass To transfer data between subroutines, functions, subprograms, and the main program. Data passes either by reference or by value. 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.

Principal value The angular representation of a complex number that falls between –112 and +112 radians.

Program unit A main program, a subroutine, a function, or a block-data subprogram.

Record A variable with a structure data type

Relocatable Not containing absolute addresses.

Row-major order The order in which array elements are stored: the rightmost subscript is incremented first when the array is mapped into contiguous memory addresses.

Run time The time during which an executable file is running.

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.

Scratch file A file created to hold temporary data, then discarded.

Segment An area of memory, less than or equal to 64K long, containing program 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.

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.

Structure A data type compounded of other data types.

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.

Truncate To convert a real number to an integer by discarding the fractional part; no rounding occurs.

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.

Union An overlaying of two or more maps; the variables in each map share the same memory locations.

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.

Visual page The area of memory whose contents currently form the graphics display.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Microsoft Product Assistance Request — FORTRAN

Microsoft Product Support Services
Phone (206) 637-7096

Instructions

When you need assistance with a Microsoft language product and you are calling from the United States, contact our Product Support Services group at **(206) 637-7096**. If you are calling from another country, please contact the nearest Microsoft subsidiary. (The subsidiaries' phone numbers are on the preaddressed labels included in the package.) So that we can answer your questions as quickly as possible, please gather all information that applies to your problem. Note or print out any on-screen messages you get when the problem occurs. Have your manual and product disks close at hand and have available all the information requested on this form when you call.

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

Diagnosing a Problem

1 Can you reproduce the problem?

❑ yes ❑ no

Steps to duplicate problem:

_____

_____

_____

_____

2 Does the problem occur with another copy of the original disk of your Microsoft software?

❑ yes ❑ no

3 Does the problem occur with another system (if available)?

❑ yes ❑ no

4 If you were running other windowing or memory-resident software at the same time, does the problem also occur when you don't use the other software?

❑ yes ❑ no

_____

Name/Version Number

_____

Name/Version Number

5 Which version of the linker are you using? (To display the version number on your screen, type LINK at the DOS or OS/2 prompt and press ENTER.)

_____

Version Number

Product

_____

Name/Version Number

Operating System

_____

Name/Version Number

Hardware

Computer

_____

Manufacturer/Model CPU (e.g., 8088, 80286)

_____

Capacity (megabyte)

Note: If using DOS, you can run CHKDSK to determine the amount of memory available. If using Apple® Macintosh® Finder™, select "About the Finder..." from the Apple menu to determine the amount of memory available.

▪ Floppy-disk drives

Number: ❑ 1 ❑ 2 ❑ other

Density: ❑ single ❑ double ❑ quad

Capacity 5.25": ❑ 160K ❑ 360K ❑ 1.2 MB

3.5": ❑ 360K ❑ 720K ❑ 1.4 MB

▪ Hard Disks

_____

Manufacturer/Model Capacity (megabyte)

_____

Manufacturer/Model

Hardware (continued)

Peripherals

- **Printer/Plotter**

Manufacturer/Model ❑ Serial ❑ Parallel

Printer peripherals, such as font cartridges, downloadable fonts, sheet feeders:

- **Mouse**

Microsoft® Mouse: ❑ Bus ❑ Serial ❑ InPort® ❑ PS/2® ❑ Other

Manufacturer/Model

- **Boards**

❑ Add-on RAM board/EMS boards

Manufacturer/Model/Total Memory

❑ Graphics-adapter board

Manufacturer/Model

❑ Other boards installed

Manufacturer/Model

Manufacturer/Model

Capacity (megabyte)

CD-ROM Player

Manufacturer/Model

Version of Microsoft MS-DOS® CD-ROM Extensions:

Network

Is your system part of a network? ❑ yes ❑ no

Manufacturer/Model

What software does your network use?

◆ ◆ ◆

Name

Address

City/State/Zip

Phone ( ) — (home) ( ) — (work)

May we contact you for additional information about your comments? Yes ____ No ____

Additional comments:

Please mail this form to:

Microsoft Corporation
One Microsoft Way
Redmond, WA
98052-6399

Attn: Languages—FORTRAN 5.1

Documentation Feedback – Microsoft® FORTRAN Version 5.1

Please help us improve our documentation. After you've become familiar with our product, please complete and return this form. Comments and suggestions become the property of Microsoft Corporation.

Which statement best describes your experience with FORTRAN?

____ I haven't had much programming experience in any language.
____ I have used other languages, but I'm new to FORTRAN.
____ I have used FORTRAN occasionally, but I'm still unfamiliar with many of its features.
____ I use FORTRAN regularly in my professional work, but I'm not a full-time programmer.
____ I'm a full-time programmer using FORTRAN regularly.

If you are not a full-time programmer, what is your primary occupation? _____

_____

Please describe the types of programs you are most likely to create with this FORTRAN product._____

_____

_____

How long ago did you buy this FORTRAN package?

____ Months

Have you read *Installing and Using* all the way through?

____ I haven't used it at all.
____ I've read part of it. Which parts?_____

_____

_____

____ I've read it all the way through.

Which statement best summarizes your response to the FORTRAN language information in *Microsoft® FORTRAN Advanced Topics*?

____ It's too simple; I need more in-depth information.
____ It's about right: I can usually understand it without much difficulty.
____ It's too technical; I find it hard to read and apply.

Do you use the Microsoft FORTRAN Advisor (online help)? ____ Yes ____ No
Why or why not? _____

_____

Normally, what percentage of the time do you compile and link

____ In one step?
____ Separately?

How often do you use the Programmer's WorkBench?

____ Not at all; prefer my own editor.
____ Occasionally.
____ Regularly.

Did you use the Programmer's WorkBench tutorial in *Environment and Tools*? ____ Yes ____ No
If you used it, was it

____ Too simple?
____ About right?
____ Too difficult?

How often do you use the Microsoft extensions to FORTRAN 77?

____ Rarely; don't need them.
____ Restrict use to assure compatibility with other compilers.
____ Regularly.

In general, did you find this documentation

____ Easy to use?
____ About average?
____ Hard to use?

Were there any topics you felt weren't covered well enough anywhere in the documentation?
Please explain. _____

_____

_____

_____

Overall, how well does the documentation in this package meet your needs? Rate each from 1 (does not meet your needs at all) to 5 (meets your need perfectly).

____ *Installing and Using* _____
____ *Reference* _____
____ *Advanced Topics* _____
____ *Environment and Tools* _____
____ *Quick Reference Guide* _____

Use the space below and on the back of this form for additional comments. Please note any errors and special ...ths or weaknesses in areas such as programming examples, indexing, and overall organization. Which parts do ...ack to most frequently?

Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft®
Making it all make sense™

0491 Part No. 21013