Fortran 77 Language Reference Manual

Introduction

This manual describes the Fortran 77 language specifications as implemented on the Silicon Graphics IRIS−4D series workstation. This implementation of Fortran 77 contains full American National Standard Institute (ANSI) Programming Language Fortran (X3.9−1978). It has extensions that provide full VMS Fortran compatibility to the extent possible without the VMS operating system or VAX data representation. It also contains extensions that provide partial compatibility with programs written in SVS Fortran and Fortran 66.

This manual refers to Fortran 77 as Fortran, except where specific distinctions between Fortran 77 and Fortran 66 are discussed.

The compiler can convert source programs written in VMS Fortran into machine programs executable under IRIX™.

Intended Audience

This manual is intended as a reference manual, rather than a tutorial, and assumes familiarity with an algebraic language or prior exposure to Fortran.

Corequisite Publications

This manual describes the Fortran language specifications. Refer to the *Fortran 77 Programmer's Guide* for information on

- How to compile and link edit a Fortran program
- Alignments, sizes, and variable ranges for the various data types
- The coding interface between Fortran programs and programs written in C and Pascal
- File formats, run−time error handling, and other information related to the IRIX operating system
- Operating system functions and subroutines callable by Fortran programs

Refer to the *IRIS−4D Series Compiler Guide* for information on:

- An overview of the compiler system
- Information on improving the program performance, showing how to use the profiling and optimization facilities of the compiler system
- The dump utilities, archiver, and other tools for maintaining Fortran programs

Refer to the *dbx User's Reference Manual* for a detailed description of the debugger (*dbx*).

For information on the interface to programs written in assembly language, refer to the *Assembly Language Programmer's Guide*.

Organization of Information

This manual contains the following chapters and appendix:

- Chapter 1, "Fortran Elements and Concepts," provides definitions for the various elements of a Fortran program.
- Chapter 2, "Constants and Data Structures,"discusses the various types of Fortran constants and explains a few ways data can be structured.
- Chapter 3, "Expressions," describes the formation, interpretation, and evaluation rules for each type of Fortran expression.
- Chapter 4, "Specification Statements," summarizes the Fortran specification statements.
- Chapter 5, "Assignment and Data Statements," discusses the types of assignment statements and explains how to use them. It also explains how to initialize variables and array elements using **DATA** statements.
- Chapter 6, "Control Statements," explains the various Fortran control statements.
- Chapter 7, "Input/Output Processing," discusses the programmer−related aspects of Fortran input/output processing.
- Chapter 8, "Input/Output Statements," describes the statements that control the transfer of data within internal storage and between internal storage and external storage devices. It also provides an overview of the Fortran input/output statements and lists the syntax, rules, and examples for each.
- Chapter 9, "Format Specification,"describes the **FORMAT** statement, field descriptors, edit descriptors, and list−directed formatting.
- Chapter 10, "Statement Functions and Subprograms," discusses user−written subprograms and explains the syntax and rules for defining program units.
- Chapter 11, "Compiler Options,"describes the options that affect source programs both during compilation and at run time.
- Appendix A, "Intrinsic Functions,"lists the intrinsic functions supported.

Typographical Conventions

The following conventions and symbols are used in the text to describe the form of Fortran statements:

Bold Indicates literal command line options, filenames, keywords, function/subroutine names, pathnames, and directory names.

Italics Represents user−defined values. Replace the item in italics with a legal value. Italics are also used for command names, manual page names, and manual titles.

`Courier` Indicates command syntax, program listings, computer output, and error messages.

`Courier bold`

Indicates user input.

[ ] Enclose optional command arguments.

()	Surround arguments or are empty if the function has no arguments following function/subroutine names. Surround manual page section in which the command is described following IRIX commands.
|	Ssparates two or more optional items.
...	Indicates that the preceding optional items can appear more than once in succession.
#	IRIX shell prompt for the superuser.
%	IRIX shell prompt for users other than superuser.

Here are two examples illustrating the syntax conventions.

DIMENSION *a*(*d*) [,*a*(*d*)] ...

indicates that the Fortran keyword **DIMENSION** must be written as shown, that the user−defined entity*a*(*d*) is required, and that one or more of *a*(*d*) can be optionally specified. Note that the pair of parentheses ( ) enclosing *d* is required.

{STATIC | AUTOMATIC} *v* [,*v*] ...

indicates that either the **STATIC** or **AUTOMATIC** keyword must be written as shown, that the user−defined entity*v* is required, and that one or more of *v* items can be optionally specified.

Chapter 1

Fortran Elements and Concepts

This chapter contains the following subsections:

- "Fortran Character Set"
- "Data Types"
- "Collating Sequence"
- "Symbolic Names"
- "Variables"
- "Source Program Lines"
- "Statements"
- "Program Units"
- "Program Organization"

This chapter provides definitions for the various elements of a Fortran program. The Fortran language is written using a specific set of characters that form the words, numbers, names, and expressions that make up Fortran statements. These statements form a Fortran program. The Fortran character set, the rules for writing Fortran statements, the main structural elements of a program, and the proper order of statements in a program are also discussed in this chapter.

Fortran Character Set

The Fortran character set consists of 26 uppercase and 26 lowercase letters (*alphabetic* characters), the numbers 0 through 9 (*digits*), and *special* characters. This manual refers to letters (uppercase and lowercase) together with the underscore (_) as *extended alphabetic* characters. The *extended alphabetic* characters together with the digits are also referred to as *alphanumeric* characters. The complete character set is

Letters: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

Digits: 0 1 2 3 4 5 6 7 8 9

Special Characters:

	Blank
=	Equal
+	Plus
–	Minus
*	Asterisk

/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal point
$	Currency symbol
'	Apostrophe
:	Colon
!	Exclamation point
_	Underscore
"	Quotation mark

Lowercase alphabetic characters, the exclamation point (!), the underscore (_), and the double quote (") are extensions to Fortran 77. Digits are interpreted in base 10. A special character can serve as an operator, a part of a character constant, a part of a numeric constant, or some other function

Blank Characters

Use blank characters freely to improve the appearance and readability of Fortran statements. They have no significance in Fortran statements, except

- in character constants
- for H and character editing in format specifications
- in Hollerith constants
- to signify an initial line when used in column 6 of source line
- when counting the total number of characters allowed in any one statement

These special considerations are discussed in detail in later sections.

Escape Sequences

Table 1−1lists escape sequences for representing non−graphic characters and for compatibility with the C programming language.

Table 1−1 C Escape Sequences

Sequence	Meaning
\n	New line
\t	Tab
\b	Backspace
\f	Form feed

\0	Null
\'	Apostrophe (does not terminate a string)
\"	Quotation mark (does not terminate a string)
\\	\
\x	x represents any character

The compiler treats the backslash character as the beginning of an escape sequence by default. To use backslash as a normal character, compile the program with the **-backslash** option.

Data Types

In general, there are three kinds of entities that have a data type: constants, data names, and function names. The types of data allowed in Fortran are

- **INTEGER**—positive and negative integral numbers and zero
- **REAL**—positive and negative numbers with a fractional part and zero
- **DOUBLE PRECISION**—same as **REAL** but using twice the storage space and possibly greater precision
- **COMPLEX**—ordered pair of **REAL** data: real and imaginary components
- **DOUBLE COMPLEX**—ordered pair of double–precision data
- **LOGICAL**—Boolean data representing *true* or *false*
- **CHARACTER**—character strings
- **HOLLERITH**—an historical data type for character definition

Together, **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, and **DOUBLE COMPLEX** constitute the class of *arithmetic* data types.

The type of data is established in one of two ways: implicitly, depending on the first letter of its symbolic name (described in this chapter), or explicitly through a type statement (described in Chapter 4). A data value can be a variable or a constant, that is, its value either can or cannot change during the execution of a program. An array is a sequence of data items occupying a set of consecutive bytes.

If not explicitly specified by a type statement or a **FUNCTION** statement, the data type of a data item, data name, or function name is determined implicitly by the first character of its symbolic name. By default, symbolic names beginning with I, J, K, L, M, or N (uppercase or lowercase) imply an **INTEGER** data type; names beginning with all other letters imply a **REAL** data type. You can change or confirm the default implicit data type corresponding to each letter of the alphabet through an **IMPLICIT** statement (refer to "EXTERNAL" of Chapter 4 for details).

The data type of external functions and statement functions is implicitly determined in the same manner as above. The type of an external function can also be explicitly declared in a **FUNCTION** statement.

Collating Sequence

The Fortran collating sequence defines the relationship between letters and digits and is used when comparing character strings. The collating sequence is determined by these rules:

- A is less than Z, and a is less than z. The listing order of the alphabetic characters specifies the collating sequence for *alphabetic* characters. The relationship between lowercase and uppercase of the same letter is unspecified.
- 0 is less than 9. The order in which digits are listed above defines the collating sequence for digits.
- Alphabetic characters and digits are not intermixed in the collating sequence.
- The blank character is less than the letter A (uppercase and lowercase) and less than the digit 0.
- The special characters given as part of the character set are *not* listed in any specific order. There is no specification as to where special characters occur in the collating sequence.

Symbolic Names

A symbolic name is a sequence of characters that refer to a memory location by describing its contents. Symbolic names identify the following user−defined local and global entities:

Local	variableconstantarraystatement functionintrinsic functiondummy procedure
Global	common blockexternal functionsubroutinemain programblock data subprogram

Conventions

A symbolic name can contain any alphanumeric character; digits and _ (underscore) are allowed in addition to uppercase and lowercase alphabetic characters. However, the first character *must* be a letter.

- Fortran symbolic names can contain any number of characters, but only the first 32 of these are significant in distinguishing one symbolic name from another (standard Fortran 77 allows only 6 characters.) Symbolic names that are used externally (program names, subroutine names, function names, common block names) are limited to 32 significant characters.
- The inclusion of the special period (.), underscore (_), and dollar sign ($) characters in symbolic names is an enhancement to Fortran 77.

Examples of valid symbolic names are

```
CASH  C3P0 R2D2  LONG_NAME
```

Examples of invalid symbolic names are

```
X*4      (contains a special character, *)
3CASH    (first character is a digit)
```

Data Types of Symbolic Names

A symbolic name has a definite data type in a program unit that can be any of the following:

- **BYTE**

- **INTEGER [*1 | *2 | *4]**
- **REAL [*4 | *8**] or **DOUBLE PRECISION**
- **COMPLEX [*8 | *16]**
- **LOGICAL [*1 | *2 | *4]**
- **CHARACTER [*** *n* **]**

The optional length specifier that follows the type name determines the number of bytes of storage for the data type. If the length specifier is omitted, the compiler uses the defaults listed in the *Fortran 77 Programmer's Guide*.

In general, wherever the usage of a given data type is allowed, it can have any internal length. One exception is the use of integer variables for assigned **GOTO** statements. In this case the integer variable must be 4 bytes in length.

Data of a given type and different internal lengths can be intermixed in expressions, and the resultant value will be the larger of the internal representations in the expression.

Note: The lengths of arguments in actual and formal parameter lists and **COMMON** blocks must agree in order produce predictable results.

Scope of Symbolic Names

The following rules determine the scope of symbolic names:

- A symbolic name that identifies a global entity, such as a common block, external function, subroutine, main program, or block data subprogram, has the scope of an executable program. Do not use it to identify another global entity in the same executable program.
- A symbolic name that identifies a local entity, such as an array, variable, constant, statement function, intrinsic function, or dummy procedure, has the scope of a single program unit. Do not use it to identify any other local entity in the same program unit.
- Do not use a symbolic name assigned to a global entity in a program unit for a local entity in the same unit. However, you can use the name for a common block name or an external function name that appears in a **FUNCTION** or **ENTRY** statement.

Variables

A variable is an entity with a name, data type, and value. Its value is either defined or undefined at any given time during program execution.

The variable name is a symbolic name of the data item and must conform to the rules given for symbolic names. The type of a variable is explicitly defined in a type–statement or implicitly by the first character of the name.

A variable cannot be used or referred to unless it has been defined through an assignment statement, input statement, **DATA** statement, or through association with a variable or array element that has been defined.

Source Program Lines

A source program line is a sequence of character positions, called *columns*, numbered consecutively starting from column 1 on the left.

Format

The two formats for Fortran programs are

- Fixed format—based on columns
- TAB format—based on the tab character

Fixed Format

A Fortran line is divided into columns, with one character per column as indicated in Table 1−2

Table 1−2 Fortran Line Structure

Field	Column
Statement label	1 through 5
Continuation indicator	6
Statement	7 to the end of the line or to the start of the comment field
Comment (optional)	73 through end of line

The **-col72**, **-col120**, **-extend_source**, and **-noextend_source** command line options are provided to change this format. See Chapter 1 of the *Fortran77 Programmer's Guide* for details. Several of these options can be specified in−line as described in Chapter 11, "Compiler Options."

TAB Character Formatting

Rather than aligning characters in specific columns, the TAB character can be used as an alternative field delimiter, as follows:

1. Type the statement label and follow it with a TAB. If there is no statement label, start the line with a TAB.
2. After the TAB, type either a statement initial line or a continuation line. A continuation line must contain a digit (1 through 9) immediately following the TAB. If any character *other* than a nonzero digit follows the TAB, the line will be interpreted as an initial line.
3. In a continuation line beginning with a TAB followed by a nonzero digit, any characters following the digit to the end of the line are a continuation of the current statement.
4. TAB−formatted lines do not have preassigned comment fields. All characters to the end of the line are considered part of the statement. However, you can use an exclamation point (!) to explicitly define a comment field. The comment field extends from the exclamation point to the end of the line.

The rules for TAB formatting can be summarized as *statement label* TAB *statement* (initial line)

TAB *continuation field statement* (continuation line)

TAB *statement* (initial line)

Note that although many terminal and text editors advance the cursor to a predetermined position when a TAB is entered, this action is not related to how the TAB will be ultimately interpreted by the compiler. The compiler interprets TABs in the statement field as blanks.

Types of Lines

The four types of Fortran program lines are

- comment
- debugging (an extension to Fortran 77)
- initial
- continuation

Comments

A comment line is used solely for documentation purposes and does not affect program execution. A comment line can appear anywhere in a program and has one of the following characteristics:

- An uppercase **C** or an asterisk (*) in column 1 and any sequence of characters from column 2 through to the end of the line
- A blank line
- Text preceded by an exclamation point (!) at any position of the line

Debugging Lines

Specify a **D** in column 1 for debugging purposes; it conditionally compiles source lines in conjunction with the **-d_lines** option described in Chapter 1 of the *Fortran 77 Programmer's Guide*. When the option is specified at compilation, all lines with a **D** in column 1 are treated as lines of source code and compiled; when the option is omitted, all lines with a **D** in column 1 are treated as comments.

Initial Lines

Initial lines contain the Fortran language statements that make up the source program; these statements are described in detail in "Program Organization". These fields divide each Fortran line into

- statement label field
- continuation indicator field
- statement field
- comment field

The fields in a Fortran line can be entered either on a character−per−column basis or by using the TAB character to delineate the fields, as described in the previous section.

Continuation Lines

A continuation line continues a Fortran statement and is identified as follows:

- Columns 1 through 5 must be blank.
- Column 6 contains any Fortran character other than a blank or the digit 0. Column 6 is frequently used to number the continuation lines.

As with initial lines, columns 7 through the end of the line contain the Fortran statement or a continuation of the statement.

Alternatively, you can use an ampersand (**&**) in column 1 to identify a continuation line. Using an **&** in column 1 implies that columns 2 through the end of the line are part of the statement. In Fortran 77, any remaining columns (column 73 and on) of a continuation line are not interpreted.

The maximum number of consecutive continuation lines is 99 unless you change this limit with the **-NC** compiler option.

Statements

Fortran statements are used to form program units. All Fortran statements, except assignment and statement functions, begin with a keyword. A *keyword* is a sequence of characters that identifies the type of Fortran statement.

A statement cannot begin on a line that contains any portion of a previous statement, except as part of a logical **IF** statement.

The **END** statement signals the physical end of a Fortran program unit and begins in column 7 or any later column of an initial line.

Statement Labels

A statement label allows you to refer to individual Fortran statements. A statement label consists of one to five digits—one of which must be nonzero—placed anywhere in columns 1 through 5 of an initial line. Blanks and leading zeros are not significant in distinguishing between statement labels.

The following statement labels are equivalent:

```
" 123 "   "123  "   "1 2 3"   "00123"
```

Each statement label must be unique within a program unit.

Fortran statements do not require labels. However, only labeled statements can be referenced by other Fortran statements. Do not label **PROGRAM**, **SUBROUTINE**, **FUNCTION**, **BLOCK DATA**, or **INCLUDE** statements.

Executable Statements

An executable statement specifies an identifiable action and is part of the execution sequence, (described in "Program Organization") in an executable program.

The three classes of executable statements are

- assignment statements
  - arithmetic
  - logical
  - statement label (**ASSIGN**)
  - character assignment
- control statements
  - unconditional, assigned, and computed **GO TO**
  - arithmetic **IF** and logical **IF**
  - block **IF**, **ELSE IF**, **ELSE**, and **END IF**
  - **CONTINUE**
  - **STOP** and **PAUSE**
  - **DO**
  - **CALL** and **RETURN**
  - **END**
- I/O statements
  - **READ**, **WRITE**, and **PRINT**
  - **REWIND**, **BACKSPACE**, **ENDFILE**, **OPEN**, **CLOSE**, and **INQUIRE**
  - **ACCEPT**, **TYPE**, **ENCODE**, **DECODE**, **DEFINE FILE**, **FIND**, **REWRITE DELETE**, and **UNLOCK**

Non−executable Statements

A non−executable statement is not part of the execution sequence. You can specify a statement label on most types of non−executable statements, but you cannot also specify that label for an executable statement in the same program unit.

A non−executable statement can perform one of these functions:

- Specify the characteristics, storage arrangement, and initial values of data
- Define statement functions
- Specify entry points within subprograms
- Contain editing or formatting information

- Classify program units
- Specify inclusion of additional statements from another source

The following data type statements are classified as non−executable:

- **CHARACTER**
- **COMPLEX**
- **DIMENSION**
- **DOUBLE PRECISION**
- **INTEGER**
- **LOGICAL**
- **REAL**
- **BYTE**

Additional non−executable program statements are

BLOCK DATA	**INCLUDE**
COMMON	**INTRINSIC**
DATA	**PARAMETER**
ENTRY	**POINTER**
EQUIVALENCE	**PROGRAM**
EXTERNAL	**SAVE**
FORMAT	**SUBROUTINE**
FUNCTION	Statement function
IMPLICIT	**VIRTUAL**

Program Units

Fortran programs consist of one or more program units. A program unit consists of a sequence of statements and optional comment lines. It can be a main program or a subprogram. The program unit defines the scope for symbolic names and statement labels.

The **END** statement must always be the last statement of a program unit.

Main Program

The main program is the program unit that initially receives control on execution. It can have a **PROGRAM** statement as its first statement and contain any Fortran statement except a**FUNCTION**, **SUBROUTINE**, **BLOCK DATA**, **ENTRY**, or **RETURN** statement. A **SAVE** statement in a main program does not affect the status of variables or arrays. A **STOP** or **END** statement in a main program terminates execution of the program.

The main program does not need to be a Fortran program. Refer to the *Fortran 77 Programmer's Guide* for information on writing Fortran programs that interact with programs written in other languages.

The main program cannot be referenced from a subprogram or from itself.

Subprograms

A subprogram is an independent section of code designed for a specialized purpose. It receives control when referenced or called by a statement in the main program or another subprogram.

A subprogram can be a

- function subprogram identified by a **FUNCTION** statement
- subroutine subprogram identified by a **SUBROUTINE** statement
- block data subprogram identified by a **BLOCK DATA** statement
- non−Fortran subprogram

Subroutines, external functions, statement functions, and intrinsic functions are collectively called procedures. A *procedure* is a program unit that performs an operational function.

An external procedure is a function or subroutine subprogram that is processed independently of the calling or referencing program unit. It can be written as a non−Fortran subprogram as described in the *Fortran 77 Programmer's Guide.*

Intrinsic functions are supplied by the processor and are generated as in−line functions or library functions. Refer to Appendix A, "Intrinsic Functions," for a description of the functions, the results given by each, and their operational conventions and restrictions.

Program Organization

This section explains the requirements for an executable Fortran program. It also describes the rules for ordering statements and the statement execution sequence.

Executable Programs

An executable program consists of exactly one main program and zero or more of each of the following entities

- function subprogram
- subroutine subprogram
- block data subprogram
- non−Fortran external procedure

The main program must not contain an **ENTRY** or a **RETURN** statement. On encountering a **RETURN** statement, the compiler issues a warning message; at execution time, a **RETURN** statement stops the program. Execution of a program normally ends when a **STOP** statement is executed in any program unit or when an **END** statement is executed in the main program.

Order of Statements

The following rules determine the order of statements in a main program or subprogram:

- In the main program, a **PROGRAM** statement is optional; if used, it must be the first statement. In other program units, a **FUNCTION**, **SUBROUTINE**, or **BLOCK DATA** statement must be the first statement.
- Comment lines can be interspersed with any statement and can precede a **PROGRAM**, **FUNCTION** , **SUBROUTINE**, or **BLOCK DATA** statement.
- **FORMAT** and **ENTRY** statements can be placed anywhere within a program unit after a **PROGRAM**, **FUNCTION**, **SUBROUTINE**, or **BLOCK DATA** statement.
- **ENTRY** statements can appear anywhere in a program unit except
  - between a block **IF** statement and its corresponding **END IF** statement
  - within the range of a **DO** loop that is, between a **DO** statement and the terminal statement of the **DO** loop
- The Fortran 77 standard requires that specification statements, including the **IMPLICIT** statement, be placed before all **DATA** statements, statement function statements, and executable statements.

  However, this implementation of Fortran permits the interspersing of **DATA** statements among specification statements.

  Specification statements specifying the type of symbolic name of a constant must appear before the **PARAMETER** statement that identifies the symbolic name with that constant.
- The Fortran 77 standard allows **PARAMETER** statements to intersperse with **IMPLICIT** statements or any other specification statements, but a **PARAMETER** statement must precede a **DATA** statement.

  This implementation extends the Fortran 77 standard to allow interspersing **DATA** statements among **PARAMETER** statements.

  PARAMETER statements that associate a symbolic name with a constant must precede all other statements containing that symbolic name.
- All statement function statements must precede the first executable statement.
- **IMPLICIT** statements must precede all other specification statements except **PARAMETER** statements.
- The last statement of a program unit must be an **END** statement.

Note: The above rules apply to the program statements after lines added by all **INCLUDE** statements are merged. **INCLUDE** statements can appear anywhere in a program unit.

Execution Sequence

The execution sequence in a Fortran program is the order in which statements are executed. Fortran statements are normally executed in the order they appear in a program unit. In general, the execution sequence is as follows:

1. Execution begins with the first executable statement in a main program and continues from there.
2. When an external procedure is referenced in a main program or in an external procedure, execution of the calling or referencing statement is suspended. Execution continues with the first executable statement in the called procedure immediately following the corresponding **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement.
3. Execution is returned to the calling statement by an explicit or implicit return statement.
4. Normal execution proceeds from where it was suspended or from an alternate point in the calling program.
5. The executable program is normally terminated when the processor executes a **STOP** statement in any program unit or an **END** statement in the main program. Execution is also automatically terminated when an operational condition prevents further processing of the program.

The normal execution sequence can be altered by a Fortran statement that causes the normal sequence to be discontinued or causes execution to resume at a different position in the program unit. Statements that cause a transfer of control are

- **GO TO**
- arithmetic **IF**
- **RETURN**
- **STOP**
- an I/O statement containing an error specifier or end−of−file specifier
- **CALL** with an alternate return specifier
- a logical **IF** containing any of the above forms
- block **IF** and **ELSE IF**
- the last statement, if any, of an **IF** block or **ELSE IF** block
- **DO**
- terminal statement of a **DO** loop
- **END**

Chapter 2

Constants and Data Structures

This chapter contains the following subsections:

- "Constants"
- "Character Substrings"
- "Records"
- "Arrays"

This chapter discusses the various types of Fortran constants and provides examples of each. It also explains a few of the ways data can be structured, including character substrings, records, and arrays.

Constants

A *constant* is a data value that cannot change during the execution of a program. It can be of the following types:

- arithmetic
- logical
- character
- Hollerith
- bit

The form in which a constant is written specifies both its value and its data type. A symbolic name can be assigned for a constant using the **PARAMETER** statement. Blank characters occurring within a constant are ignored by the processor unless the blanks are part of a character constant.

The sections that follow describe the various types of constants in detail.

Arithmetic Constants

The Fortran compiler supports the following types of arithmetic constants:

- integer
- real
- double−precision
- complex

An arithmetic constant can be signed or unsigned. A signed constant has a leading plus or minus sign to denote a positive or negative number. A constant that can be either signed or unsigned is called an optionally signed constant. Only arithmetic constants can be optionally signed.

Note: The value zero is neither positive nor negative; a signed zero has the same value as an unsigned zero.

Integer Constants

An integer constant is a whole number with no decimal points; it can have a positive, negative, or zero value. Hexadecimal and octal integer constants are extensions to the standard integer constant.

Format for Integer Constants

The format for an integer constant is

sww

where

s is the sign of the number: - for negative, + (optional) for positive.

ww is a whole number.

In Fortran, integer constants must comply with the following rules:

- It must be a whole number, that is, without a fractional part.
- If negative, the special character minus (-) must be the leading character. The plus sign (+) in front of positive integers is optional.
- It must not contain embedded commas.

Examples of valid integer constants are

```
0 +0   +176   -1352   06310   35
```

Examples of invalid integer constants are

`2.03` Decimal point not allowed. This is a *real constant* (described later in this chapter).

`7,909` Embedded commas not allowed.

Hexadecimal Integer Constants

Use hexadecimal integer constants for a base 16 radix. Specify a dollar sign ($) as the first character, followed by any digit (0 through 9) or the letters A through F (either uppercase or lowercase). The following are valid examples of hexadecimal integer constants:

```
$0123456789
$ABCDEF
$A2B2C3D4
```

You can use hexadecimal integer constants wherever integer constants are allowed. Note that in mixed−mode expressions, the compiler converts these constants from type integer to the dominant type of expression in which they appear.

Octal Integer Constants

Use octal integer constants for a base 8 radix. The type of an octal integer constant is **INTEGER**, in contrast to the octal constant described in "Bit Constants". This constant is supported to provide compatibility with PDP-11 Fortran.

The format of an octal constant is as follows:

o"*string*"

where *string* is one or more digits in the range of 0 through 7.

Real Constants

A real constant is a number containing a decimal point, exponent, or both; it can have a positive, negative, or zero value.

A real constant can have the following forms:

sww.ff	Basic real constant
*sww.ff***E***see*	Basic real constant followed by a real exponent
*sww***E***see*	Integer constant followed by a real exponent

where

s	is the sign of the number: - for negative, + (optional) for positive.
ww	is a string of digits denoting the whole number part, if any.
.	is a decimal point.
ff	is a string of digits denoting the fractional part, if any.
E*see*	denotes a real exponent, where *see* is an optionally signed integer.

A basic real constant is written as an optional sign followed by a string of decimal digits containing an optional decimal point. There must be at least one digit.

A real exponent is a power of ten.

The value of a real constant is either the basic real constant or, for the forms *sww.ff***E***see* and *sww***E***see*, the product of the basic real constant or integer constant and the power of ten indicated by the exponent following the letter **E**.

All three forms can contain more digits than the precision used by the processor to approximate the value of the real constant. See the *Fortran 77 Programmer's Guide* for information on the magnitude and precision of a real number.

Table 2−1illustrates real constants written in common and scientific notation with their corresponding **E** format.

Table 2−1 Notation Forms for Real Constants

Common Notation	Scientific Notation	Real Exponent Form
5.0	0.5*10	.5E1 or 0.5E1

364.5	3.465*102	.3645E3
49,300	4.93*104	.493E5
-27,100	-2.71*104	-.271E5
-.0018	-1.8*10-3	-.18E-2

The following real constants are equivalent:

```
5E4    5.E4    .5E5    5.0E+4    +5E04    50000.
```

Table 2−2lists examples of invalid real constants and the reasons they are invalid.

Table 2−2 Invalid Real Constants

Invalid Constant	Reason Invalid
-18.3E	No exponent following the E
E-5	Exponent part alone
6.01E2.5	Exponent part must be an integer
3.5E4E2	Only one exponent part allowed per constant
19,850	Embedded commas not allowed

Double−Precision Constants

A double−precision constant is similar to a real constant except that it can retain more digits of the precision than a real constant. (The size and value ranges of double−precision constants are given in the *Fortran 77 Programmer's Guide.*)

A double−precision constant assumes a positive, negative, or zero value in one of the following forms:

*sww***D***see* An integer constant followed by a double−precision exponent

*sww.ff***D***see* A basic real constant followed by a double−precision exponent

where

s is an optional sign.

ww is a string of digits denoting the whole number part, if any.

ff is a string of digits denoting the fractional part, if any.

D*see* denotes a double−precision exponent where*see* is an optionally signed exponent.

The value of a double−precision constant is the product of the basic real constant part or integer constant part and the power of ten indicated by the integer following the letter **D** in the exponent part. Both forms can contain more digits than those used by the processor to approximate the value of the real constant. Refer to the *Fortran 77 Programmer's Guide* for information on the magnitude and precision of a double−precision constant.

Valid forms of double−precision constants are

```
1.23456D3
8.9743D0
```

```
-4.D-10
16.8D-6
```

For example, the following forms of the numeric value 500 are equivalent:

```
5D2  +5D02  5.D2  5.D+02  5D0002
```

Table 2−3lists examples of invalid double−precision constants and the reasons they are invalid.

Table 2−3 Invalid Double−Precision Constants

Invalid Constant	Reason Invalid
2.395D	No exponent following the D
-9.8736	Missing D exponent designator
1,010,203D0	Embedded commas not allowed

Complex Constants

A complex constant is a processor approximation of the value of a complex number. It is represented as an ordered pair of real data values. The first value represents the real part of the complex number, and the second represents the imaginary part. Each part has the same precision and range of allowed values as real data.

A complex constant has the form (*m, n*) where *m* and *n* each have the form of a *real constant*, representing the complex value $m + n\mathbf{i}$, where **i** is the square root of -1. The form *m* denotes the real part; *n* denotes the imaginary part. Both *m* and *n* can be positive, negative, or zero. Refer to Table 2−4for examples of valid forms of complex data.

Table 2−4 Valid Forms of Complex Data

Valid Complex Constant	Equivalent Mathematical Expression
(3.5, -5)	3.5 -5i
(0, -1)	- i
(0.0, 12)	0 + 12i or 12i
(2E3, 0)	2000 + 0i or 2000

Table 2−5provides examples of invalid constants and lists the reasons they are invalid.

Table 2−5 Invalid Forms of Complex Data

Invalid Constant	Reason Invalid
(1,)	No imaginary part
(1, 2.2, 3)	More than two parts
(10, 52.D5)	Double−precision constants not allowed for either part
(1.15, 4E)	Imaginary part has invalid form

Logical Constants

Logical constants represent only the values true or false, represented by one of the forms inTable 2−6

Table 2−6 Logical Constant Values

Form	Value

.TRUE.	True
.FALSE.	False

Character Constants

A character constant is a string of one or more characters that can be represented by the processor. Each character in the string is numbered consecutively from left to right beginning with 1.

Note: The quotation mark (") is an extension to Fortran 77.

If the delimiter is ", then a quotation mark within the character string is represented by two consecutive quotation marks with no intervening blanks.

If the delimiter is ’, then an apostrophe within the character string is represented by two consecutive apostrophes with no intervening blanks.

Blanks within the string of characters are significant.

The length of a character constant is the number of characters, including blanks, between the delimiters. The delimiters are not counted, and each pair of apostrophes or quotation marks between the delimiters counts as a single character.

A character constant is normally associated with the **CHARACTER** data type. The Fortran 77 standard is extended (except as noted below) to allow character constants to appear in the same context as a numeric constant. A character constant in the context of a numeric constant is treated the same as a Hollerith constant.

Note: Character constants cannot be used as actual arguments to numeric typed dummy arguments.

Table 2−7provides examples of valid character constants and shows how they are stored.

Table 2−7 Valid Character Constants

Constant	Stored as
’DON’’T’	DON’T
"I’M HERE!"	I’M HERE!
’STRING’	STRING
’LMN""OP’	LMN""OP

Table 2−8lists examples of invalid character constants and the reasons they are invalid.

Table 2−8 Invalid Character Constants

Invalid Constant	Reason Invalid
’ISN.T	Terminating delimiter missing
.YES’	Mismatched delimiters
CENTS	Not enclosed in delimiters
’’	Zero length not allowed
""	Zero length not allowed

Hollerith Constants

Use Hollerith constants to manipulate packed character strings in the context of integer data types. A Hollerith constant consists of a character count followed by the letter **H** (either uppercase or lowercase) and a string of characters as specified in the character count and has the following format:

*n***H***xxx...x*

where *n* is a nonzero, unsigned integer constant and where the *x*'s represent a string of exactly *n* contiguous characters. The blank character is significant in a Hollerith constant.

Examples of valid Hollerith constants are

```
3H A
10H'VALUE = '
8H MANUAL
```

Table 2–9provides some examples of invalid Hollerith constants and the reasons they are invalid.

Table 2–9 Invalid Hollerith Constants

Invalid Constant	Reason Invalid
2H YZ	Blanks are significant; should be 3H YZ
-4HBEST	Negative length not allowed
0H	Zero length not allowed

The following rules apply to Hollerith constants:

- Hollerith constants are stored as byte strings; each byte is the ASCII representation of one character.
- Hollerith constants have no type; they assume a numeric data type and size depending on the context in which they are used.
- When used with a a binary operator, octal and hexadecimal constants assume the data type of the other operand. For example,

  ```
  INTEGER*2 HILO
  HILO = 'FF'X
  ```

- The constant is assumed to be of type **INTEGER*2** and two bytes long.
- In other cases, when used in statements that require a specific data type, the constant is assumed to be the required type and length.
- A length of four bytes is assumed for hexadecimal and octal constants used as arguments; no data type is assumed.
- In other cases, the constant is assumed to be of type **INTEGER*4**.
- When a Hollerith constant is used in an actual parameter list of a subprogram reference, the formal parameter declaration within that subprogram must specify a numeric type, not a character type.
- A variable can be defined with a Hollerith value through a **DATA** statement, an assignment statement, or a **READ** statement.
- The number of characters (*n*) in the Hollerith constant must be less than or equal to *g*, the maximum

number of characters that can be stored in a variable of the given type, where *g* is the size of the variable expressed in bytes. If $n < g$, the Hollerith constant is stored and extended on the right with $(g - n)$ blank characters. (Refer to the *Fortran 77 Programmer's Guide* for the sizes of arithmetic and logical data types.)

Bit Constants

You can use bit constants anywhere numeric constants are allowed. Table 2–10shows the allowable bit constants and their format.

Table 2–10 Valid Substring Examples

Format	Meaning	Valid *substring* Characters	Maximum
b' *string*' or '*string*'b[a]	Binary	0, 1	64
O' *string*' or '*string*'oa	Octal	0 - 7	22
x' *string*' or '*string*'xa	Hexadecimal	0 - 9; a - f	16
z' *string*' or '*string*'za	Hexadecimal	0 - 9; a - f	16

[a] b, o, x, and z can be lower– or uppercase (B, O, X, Z)

The following are examples of bit constants used in a **DATA** statement.

```
integer a(4)
data a/b'1010',o'12',z'a',x'b'/
```

The above statement initializes the first elements of a four–element array to binary, the second element to an octal value, and the last two elements to hexadecimal values.

The following rules apply to bit constants:

- Bit constants have no type; they assume a numeric data type and size depending on the context in which they are used.
- When used with a binary operator, octal and hexadecimal constants assume the data type of the other operand. For example,

  ```
  INTEGER*2 HILO
  HILO = 'FF'X
  ```

  The constant is assumed to be of the **INTEGER*2** type and two bytes long.
- In other cases, when used in statements that require a specific data type, the constant is assumed to be the required type and length.
- A length of four bytes is assumed for hexadecimal and octal constants used as arguments; no data type is assumed.
- In other cases, the constant is assumed to be of the **INTEGER*4** data type.
- A hexadecimal or octal constant can specify up to 16 bytes of data.
- Constants are padded with zeros to the left when the assumed length of the constant is more than the digits specified by the constant. Constants are truncated to the left when the assumed length is less

than that of the digits specified.

Character Substrings

A character substring is a contiguous sequence of *characters* that is part of a character data item. A character substring cannot be empty; that is, it must contain at least one byte of storage. Each character is individually defined or undefined at any given time during the execution of a program.

Substring Names

A substring name defines the corresponding substring and allows it to be referenced in a character expression. A substring name has one of the following forms:

v([*e1*]:[*e2*])

a(*s*[,*s*]...) ([*e1*]:[*e2*])

where

v	is a character variable name.
a	is a character array name.
e1 and *e2*	are integer expressions, called substring expressions.You can specify a non−integer character for *e1* and *e2*. If specified, each non−integer character is converted to an integer before use; fractional portions remaining after conversion are truncated.
s	is a subscript expression.

The value *e1* specifies the left most character position of the substring relative to the beginning of the variable or array element from which it was abstracted, while *e2* is the right most position. Positions are numbered left to right beginning with 1. For example, the following denotes characters in positions three through five of the character variable **EX**:

```
EX(3:5)
```

The following specifies characters in positions one through five of the character array element **NAME(2,4)**:

```
NAME(2,4)(1:5)
```

A character substring has the length *e2* - *e1* + 1.

Substring Values e1, e2

The value of the numeric expressions *e1* and *e2* in a substring name must fall within the range

1 \xb2 *e1* \xb2 *e2* \xb2 *len*

where *len* is the length of the character variable or array element. A value of one is implied if *e1* is omitted. A value of *len* is taken if *e2* is omitted. When both *e1* and *e2* are not specified, the form *v*(:) is equivalent to *v* and the form *a*(*s* [,*s*]...)(:) is equivalent to *a*(*s* [,*s*]...).

The specification for *e1* and *e2* can be any numeric integer expression, including array element references and function references. Consider the character variable

```
XCHAR = 'QRSTUVWXYZ'
```

Table 2–11lists examples of valid substrings taken from this variable.

Table 2–11 Valid Substring Examples

Expression	Substring Value	Substring Length
EX1 = XCHAR (3:8)	STUVWX	6
EX2 = XCHAR (:8)	QRSTUVWX	8
EX3 = XCHAR (5:)	UVWXYZ	6

Other examples are

`BQ(10)(2:IX)` Specifies characters in positions 2 through integer **IX** of character array **BQ(10)**. The value of **IX** must be \xb3 2 and \xb2 the length of an element of **BQ**.

`BLT(:)` Equivalent to the variable **BLT**.

Records

The record–handling extension enables you to declare and operate on multifield records in Fortran programs. Avoid confusing the term *record* as it is used here with the term *record* that describes input and output data records.

Overview of Records and Structures

A *record* is a composite or aggregate entity containing one or more record elements or fields. Each element of a record is usually named. References to a record element consist of the name of the record and the name of the desired element. Records allow you to organize heterogeneous data elements within one structure and to operate on them either individually or collectively. Because they can be composed of heterogeneous data elements, records are not typed like arrays are.

Define the form of a record with a group of statements called a *structure definitionblock*. Establish a structure declaration in memory by specifying the name of the structure in a **RECORD** statement. A structure declaration block can include one or more of the following items:

- Typed data declarations (variables or arrays)
- Substructure declarations
- Mapped field declarations
- Unnamed fields

The following sections describe these items. Refer to the **RECORD** and **STRUCTURE** declarations block sections in Chapter 4, "Specification Statements," for details on specifying a structure in a source program.

Typed Data Declarations (Variables or Arrays)

Typed data declarations in structure declarations have the form of normal Fortran typed data declarations. You can freely intermix different types of data items within a structure declaration.

Substructure Declarations

Establish substructures within a structure by using either a nested structure declaration or a **RECORD** statement.

Mapped Field Declarations

Mapped field declarations are made up of one or more typed data declarations, substructure declarations (structure declarations and **RECORD** statements), or other mapped field declarations. A block of statements, called a union declaration, defines mapped field declarations. Unlike typed data declarations, all mapped field declarations that are made within a single union declaration share a common location within the containing structure.

Unnamed Fields

Declare unnamed fields in a structure by specifying the pseudo–name **%FILL** in place of an actual field name. **%FILL** generates empty space in a record for purposes such as alignment.

Record and Field References

The generic term *scalar reference* refers to all references that resolve to single typed data items. A scalar field reference of an aggregate falls into this category. The generic term *aggregate reference* is used to refer to all references that resolve to references of structured data items defined by a **RECORD** statement.

Scalar field references can appear wherever normal variables or array elements can appear, with the exception of **COMMON**, **SAVE**, **NAMELIST**, and **EQUIVALENCE** statements. Aggregate references can only appear in aggregate assignment statements, in unformatted I/O statements, and as parameters to subprograms.

Aggregate Assignment Statement

Aggregates can be assigned as whole entities. This special form of the assignment statement is indicated by an aggregate reference on the left–hand side of an assignment statement and requires an identical aggregate to appear on the right–hand side of the assignment.

Arrays

An array is a non–empty sequence of data of the same type occupying consecutive bytes in storage. A member of this sequence of data is referred to as an array element.

Each array has the following characteristics:

- array name
- data type

- array elements
- array declarator specifying:
  - number of dimensions
  - size and bounds of each dimension

Define an array using a **DIMENSION**, **COMMON**, or type statement (described in Chapter 4, "Specification Statements"); it can have a maximum of seven dimensions.

Note: For information on array handling when interacting with programs written in another language, see the *Fortran 77 Programmer's Guide.*

Array Names and Types

An array name is the symbolic name given to the array and must conform to the rules given in Chapter 1, "Fortran Elements and Concepts," for symbolic names. When referencing the array as a whole, specify only the array name. An array name is local to a program unit.

An array element is specified by the array name and a subscript. The form of an array element name is

a (*s* [,*s*]...)

where

a	is an array name.
(*s* [,*s*]...)	is a subscript.
s	is a subscript expression.

For example, **DATE(1,5)** accesses the element in the first row, fifth column, of the **DATE** array.

The number of subscript expressions must be equal to the number of dimensions in the array declarator for the array name.

An array element can be any of the valid Fortran data types. All array elements must be the same data type. Specify the data type explicitly using a type statement or implicitly using the first character of the array name. Refer to Chapter 1, "Fortran Elements and Concepts," for details about data types.

Reference a different array element by changing the subscript value of the array element name.

Array Declarators

An array declarator specifies a symbolic name for the array, the number of dimensions in the array, and the size and bounds of each dimension. Only one array declarator for an array name is allowed in a program unit. The array declarator can appear in a **DIMENSION** statement, a type statement, or a **COMMON** statement but not in more than one of these.

An array declarator has the form

a (*d* [,*d*]...)

where

a is a symbolic name of the array.

d is a dimension declarator of the following form:

[*d1*:] *d2*

where:

d1 is a lower−dimension bound that must be a numeric expression.

d2 is an upper−dimension bound that must be a numeric expression or an asterisk (*). Specify an asterisk only if *d2* is part of the last dimension declarator (see below).

If *d1* or *d2* is not of type integer, it is converted to integer values; any fractional part is truncated.

An array declarator can have a dummy argument as an array name and, therefore, be a dummy array declarator. An array declarator can be one of three types: a constant array declarator, an adjustable array declarator, or an assumed−size array declarator.

Each of the dimension bounds in a *constant array declarator* is a numeric constant expression. An *adjustable array declarator* is a dummy array declarator that contains one or more dimension bounds that are integer expressions but not constant integer expressions. An *assumed−size array declarator* is a dummy array declarator that has integer expressions for all dimension bounds, except that the upper dimension bound, *d2*, of the last dimension is an asterisk (*).

A dimension−bound expression cannot contain a function or array element name reference.

Value of Dimension Bounds

The lower−dimension bound, *d1*, and the upper−dimension bound, *d2*, can have positive, negative, or zero values. The value of the upper−dimension bound, *d2*, must be greater than or equal to that of the lower−dimension bound, *d1*.

If a lower−dimension bound is not specified, its value is assumed to be one (1). The upper−dimension bound of an asterisk (*) is always greater than or equal to the lower dimension bound.

The size of a dimension that does not have an asterisk (*) as its upper bound has the value $(d1 - d2) + 1$.

The size of a dimension that has an asterisk (*) as its upper bound is not specified.

Array Size

The size of an array is exactly equal to the number of elements contained by the array. Therefore, the size of an array equals the product of the dimensions of the array. For constant and adjustable arrays, the size is straightforward. For assumed−size dummy arrays, however, the size depends on the actual argument corresponding to the dummy array. There are three cases:

- If the actual argument is a non−character array name, the size of the assumed−size array equals the size of the actual argument array.
- If the actual argument is a non−character array element name with a subscript value of *j* in an array of size *x*, the size of the assumed−size array equals $x - j + 1$.

- If the actual argument is either a character array name, a character array element name, or a character array element substring name, the array begins at character storage unit *t* of an array containing a total of *c* character storage units; the size of the assumed−size array equals:

  INT((c - t + 1)/*ln***)**

  where *ln* is the length of an element of the dummy array.

Note: Given an assumed−size dummy array with *n* dimensions, the product of the sizes of the first n - 1 dimensions must not be greater than the size of the array (the size of the array is determined as described above).

Storage and Element Ordering

Storage for an array is allocated in the program unit in which it is declared, except in subprograms where the array name is specified as a dummy argument. The former declaration is called an actual array declaration. The declaration of an array in a subprogram where the array name is a dummy argument is called a *dummy array declaration.*

The elements of an array are ordered in sequence and stored in column order. This means that the left most subscript varies first, as compared to row order, in which the right most subscript varies first. The first element of the array has a *subscript value* of one; the second element has a *subscript value* of two; and so on. The last element has a *subscript value* equal to the size of the array.

Consider the following statement that declares an array with an **INTEGER** type statement:

```
INTEGER t(2,3)
```

Figure 2−1shows the ordering of elements of this array.

t(1,1)	t(2,1)	t(1,2)	t(2,2)	t(1,3)	t(2,3)

Figure 2−1 Order of Array Elements

Subscripts

The *subscript* describes the position of the element in an array and allows that array element to be defined or referenced. The form of a subscript is

(*s* [,*s*]...)

where *s* is a *subscript expression.* The term *subscript* includes the parentheses that delimit the list of subscript expressions.

A *subscript expression* must be a numeric expression and can contain array element references and function references. However, it cannot contain any function references that affect other subscript expressions in the same subscript.

A non−integer character can be specified for subscript expression. If specified, the non−integer character is converted to an integer before use; fractional portions remaining after conversion are truncated.

If a subscript expression is not of type integer, it is converted to integer values; any fractional part is truncated.

Because an array is stored as a sequence in memory, the values of the subscript expressions must be combined into a single value that is used as the offset into the sequence in memory. That single value is called the *subscript value.*

The subscript value determines which element of the array is accessed. The subscript value is calculated from the values of all the subscript expressions and the declared dimensions of the array (see Table 2–12).

Table 2–12 Determining Subscript Values

n	Dimension Declarator	Subscript	Subscript Value
1	(j1:k1)	(s1)	1 + (s1 - j1)
2	(j1:k1, j2:k2)	(s1, s2)	1 + (s1 - j1) + (s2 - j2)*d1
3	(j1:k1, j2:k2, j3:k3)	(s1, s2, s3)	1 + (s1-j1) + (s2-j2) * d1 + (s3-j3) * d2 * d1
n	(j1:k1, ...jn:kn)	(s1, ...sn)	1 + (s1 - j1) + (s2 - j2)*d1 + (s3-j3)*d1*d2 + ... + (sn - jn) * dn-1*dn-2*d1

The subscript value and the subscript expression value are not necessarily the same, even for a one–dimensional array. For example,

```
DIMENSION X(10,10),Y(-1:8)
Y(2) = X(1,2)
```

Y(2) identifies the fourth element of array **Y**, the subscript is **(2)** with a subscript value of four, and the subscript expression is **2** with a value of two. **X(1,2)** identifies the eleventh element of **X**, the subscript is **(1,2)** with a subscript value of eleven, and the subscript expressions are **1** and **2** with the values of one and two, respectively.

Chapter 3

Expressions

This chapter contains the following subsections:

- "Arithmetic Expressions"
- "Character Expressions"
- "Relational Expressions"
- "Logical Expressions"
- "Evaluating Expressions in General"

An expression performs a specified type of computation. It is composed of a sequence of operands, operators, and parentheses. The types of Fortran expressions are

- arithmetic
- character
- relational
- logical

This chapter describes formation, interpretation, and evaluation rules for each type of expression. This chapter also discusses mixed–mode expressions, which are Fortran 77 enhancements of Fortran 66.

Arithmetic Expressions

An arithmetic expression specifies a numeric computation that yields a numeric value on evaluation. The simplest form of an arithmetic expression can be:

- an unsigned arithmetic constant
- a symbolic name of an arithmetic constant
- an arithmetic variable reference
- an arithmetic array element reference
- an arithmetic function reference

You can form more complicated arithmetic expressions from one or more operands together with arithmetic operators and parentheses.

An arithmetic element can include logical entities because logical data is treated as integer data when used in an arithmetic context. When both arithmetic and logical operands exist for a given operator, the logical operand is promoted to type **INTEGER** of the same byte length as the original logical length. For example, a **LOGICAL*2** will be promoted to **INTEGER*2** and a **LOGICAL*4** will be promoted to **INTEGER*4**.

Arithmetic Operators

Table 3−1shows the arithmetic operators.

Table 3−1 Arithmetic Operators

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or identity
-	Subtraction or negation

Use the exponentiation, division, and multiplication operators between exactly two operands. You can use the addition and subtraction operators with one or two operands; in the latter case, specify the operator before the operand; for example, **-TOTAL**.

Do not specify two operators in succession. (Note that the exponentiation operator consists of the two characters (**), but is a *single* operator.) Implied operators, as in implied multiplication, are not allowed.

Interpretation of Arithmetic Expressions

Table 3−2interprets sample arithmetic expressions.

Table 3−2 Interpretation of Arithmetic Expressions

Operator	Use	Interpretation
**	x1 ** x2	Exponentiate x1 to the power of x2
*	x1 * x2	Multiply x1 and x2
/	x1 / x2	Divide x1 by x2
+	x1 + x2	Add x1 and x2
	+ x	x (identity)
-	x1 - x2	Subtract x1 from x2
	−x	Negate x

An arithmetic expression containing two or more operators is interpreted based on a precedence relation among the arithmetic operators. This precedence, from highest to lowest, is

- ( )
- **
- * and /
- + and -

Use parentheses to override the order of precedence.

The following is an example of an arithmetic expression:

```
A/B−C**D
```

The operators are executed in the following sequence:

1. **C**D** is evaluated first.
2. **A/B** is evaluated next.
3. The result of **C**D** is subtracted from the result of **A/B** to give the final result.

A unary operator (-) can follow another operator. Specifying the unary operator after the exponentiation operator produces a variation on the standard order of operations. The unary operator is evaluated first in that case, resulting in exponentiation taking a lower precedence in the expression.

For example, the following expression

```
A ** – B * C
```

is interpreted as

```
A ** ( – B * C )
```

Arithmetic Operands

Arithmetic operands must specify values with integer, real, double–precision, complex, or double–complex data types. You can combine specific operands in an arithmetic expression. The arithmetic operands, in order of increasing complexity, are

- primary
- factor
- term
- arithmetic expression

A *primary* is the basic component in an arithmetic expression. The forms of a primary are

- an unsigned arithmetic constant
- a symbolic name of an arithmetic constant
- an arithmetic variable reference
- an arithmetic array element reference
- an arithmetic function reference
- an arithmetic expression enclosed in parentheses

A *factor* consists of one or more primaries separated by the exponentiation operator. The forms of a factor are

- primary
- primary ** factor

Factors with more than one exponentiation operator are interpreted from right to left. For example, **I**J**K** is interpreted as **I**(J**K)**, and **I**J**K**L** is interpreted as **I**(J**(K**L))**.

The *term* incorporates the multiplicative operators into arithmetic expressions. Its forms are

- factor
- term/factor
- term * factor

The above definition indicates that factors are combined from left to right in a term containing two or more multiplication or division operators.

Finally, at the highest level of the hierarchy, are the *arithmetic expressions.* The forms of an arithmetic expression are

- term
- + term
- - term
- arithmetic expression + term
- arithmetic expression - term

An arithmetic expression consists of one or more terms separated by an addition operator or a subtraction operator. The terms are combined from left to right. For example, **A+B−C**has the same interpretation as the expression **(A+B)−C** Expressions such as **A*−B**and **A+−B**are not allowed. The correct forms are **A*(-B)** and **A+(−B)**

An arithmetic expression can begin with a plus or minus sign.

Arithmetic Constant Expressions

An *arithmetic constant expression* is an arithmetic expression containing no variables. Therefore, each primary in an arithmetic constant expression must be one of the following:

- arithmetic constant
- symbolic name of an arithmetic constant
- arithmetic constant expression enclosed in parentheses

In an arithmetic constant expression, do not specify the exponentiation operator unless the exponent is of type integer. Variable, array element, and function references are not allowed. Examples of integer constant expressions are

```
 7
-7
-7+5
3**2
x+3 (where x is the symbolic name of a constant)
```

Integer Constant Expressions

An *integer constant expression* is an arithmetic constant expression containing only integers. It can contain constants or symbolic names of constants, provided they are of type integer. As with all constant expressions, no variables, array elements, or function references are allowed.

Evaluating Arithmetic Expressions

The data type of an expression is determined by the data types of the operands and functions that are referenced. Thus, integer expressions, real expressions, double−precision expressions, complex expressions, and double expressions have values of type integer, real, double−precision, complex, and double−complex, respectively.

Single−Mode Expressions

Single−mode expressions are arithmetic expressions in which all operands have the same data type. The data type of the value of a single−mode expression is thus the same as the data type of the operands. When the addition operator or the subtraction operator is used with a single operand, the data type of the resulting expression is the same as the data type of the operand.

Mixed−Mode Expressions

Mixed−mode expressions contain operands with two or more data types. The data type of the result of a mixed−mode expression depends on the rank associated with each data type, as shown inTable 3−3

Table 3−3 Data Type Ranks

Data Type	Rank
INTEGER*1	1 (lowest)
INTEGER*2	2
INTEGER*4	3
REAL*4	4
REAL*8 (double precision)	5
COMPLEX*8	6
COMPLEX*16	7 (highest)

Except for exponentiation (discussed below), the result of a mixed−mode expression is assigned the data type of the highest−ranked element in the expression. The lower−ranked operand is converted to the type of the higher−ranked operand so that the operation is performed on values with equivalent data types. For example, an operation on an integer operand and a real operand produces a result of type real.

Operations that combine **REAL*8** (**DOUBLE PRECISION**) and **COMPLEX*8** (**COMPLEX**) are not allowed. The **REAL*8** operand must be explicitly converted (for example, by using the **SNGL** intrinsic function).

Exponentiation

Exponentiation is an exception to the above rules for mixed−mode expressions. When raising a value to an integer power, the integer is not converted. The result is assigned the type of the left operand.

When a complex value is raised to a complex power, the value of the expression is defined as follows:

```
xy = EXP (y * LOG(x))
```

Integer Division

One operand of type integer can be divided by another operand of type integer. The result of an integer division operation is a value of type integer, referred to as an integer quotient. The integer quotient is obtained as follows:

- If the magnitude of the mathematical quotient is less than one, then the integer quotient is zero. For example, the value of the expression (18/30) is zero.
- If the magnitude of the mathematical quotient is greater than or equal to one, then the integer quotient is the largest integer that does not exceed the magnitude of the mathematical quotient and whose sign is the same as that of the mathematical quotient. For example, the value of the expression (-9/2) is -4.

Character Expressions

A *character expression* yields a character string value on evaluation. The simplest form of a character expression can be one of these types of characters:

- constant
- variable reference
- array element reference
- substring reference
- function reference

Construct complicated character expressions from one or more operands together with the concatenate operator and parentheses.

Concatenate Operator

The concatenate operator (//) is the only character operator defined in Fortran. A character expression formed from the concatenation of two character operands *x1* and *x2* is specified as

x1 // *x2*

The result of this operation is a character string with a value of *x1* extended on the right with the value of *x2*. The length of the result is the sum of the lengths of the character operands. For example,

```
'HEL' // 'LO2'
```

The result of the above expression is the string **HELLO2** of length six.

Character Operands

A *character operand* must identify a value of type character and must be a character expression. The basic component in a character expression is the character *primary*. The forms of a character primary are

- character constant
- symbolic name of a character constant
- character variable reference
- character array element reference
- character substring reference
- character function reference
- character expression enclosed in parentheses

A *character expression* consists of one or more character primaries separated by the concatenation operator. Its forms are

- character primary
- character expression // character primary

In a character expression containing two or more concatenation operators, the primaries are combined from left to right. Thus, the character expression

```
'A' // 'BCD' // 'EF'
```

is interpreted the same as

```
('A' // 'BCD') // 'EF'
```

The value of the above character expression is **ABCDEF**.

Except in a character assignment statement, concatenation of an operand with an asterisk (*) as its length specification is not allowed unless the operand is the symbolic name of a constant.

Character Constant Expressions

A *character constant* expression is made up of operands that cannot vary. Each primary in a character constant expression must be a

- character constant
- symbolic name of a character constant
- character constant expression enclosed in parentheses

A character constant expression cannot contain variable, array element, substring, or function references.

Relational Expressions

A relational expression yields a logical value of either **.TRUE.** or **.FALSE.** on evaluation and comparison of two arithmetic expressions or two character expressions. A relational expression can appear only within a logical expression. Refer to "Logical Expressions" for details about logical expressions.

Relational Operators

Table 3–4lists the Fortran relational operators.

Table 3–4 Fortran Relational Operators

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

Arithmetic and character operators are evaluated *before* relational operators.

Relational Operands

The operands of a relational operator can be arithmetic or character expressions. The relational expression requires exactly two operands and is written in the following form:

e1 relop e2

where

e1 and *e2* are arithmetic or character expressions.

relop is the relational operator.

Note: Both *e1* and *e2* must be the same type of expression, either arithmetic or character.

Evaluating Relational Expressions

The result of a relational expression is of type logical, with a value of **.TRUE.** or **.FALSE.**. The manner in which the expression is evaluated depends on the data type of the operands.

Arithmetic Relational Expressions

In an arithmetic relational expression, *e1* and *e2* must each be an integer, real, double precision, complex, or double complex expression. *relop* must be a relational operator.

The following are examples of arithmetic relational expressions:

```
(a + b) .EQ. (c + 1)
HOURS .LE. 40
```

You can use complex type operands only when specifying either the **.EQ.** or **.NE.** relational operator.

An arithmetic relational expression has the logical value **.TRUE.** only if the values of the operands satisfy the relation specified by the operator. Otherwise, the value is **.FALSE.**.

If the two arithmetic expressions *e1* and *e2* differ in type, the expression is evaluated as follows:

```
((e1) – (e2)) relop 0
```

where the value 0 (zero) is of the same type as the expression ((*e1*)– (*e2*)) and the type conversion rules apply to the expression. Do not compare a double precision value with a complex value.

Character Relational Expressions

In a character relational expression, *e1* and *e2* are character expressions and *relop* is a relational operator.

The following is an example of a character relational expression:

```
NAME .EQ. 'HOMER'
```

A character relational expression has the logical value **.TRUE.** only if the values of the operands satisfy the relation specified by the operator. Otherwise, the value is **.FALSE.**. The result of a character relational expression depends on the collating sequence as follows:

- If *e1* and *e2* are single characters, their relationship in the collating sequence determines the value of the operator. *e1* is less than or greater than *e2* if *e1* is before or after *e2*, respectively, in the collating sequence.
- If either *e1* or *e2* are character strings with lengths greater than 1, corresponding individual characters are compared from left to right until a relationship other than **.EQ.** can be determined.
- If the operands are of unequal length, the shorter operand is extended on the right with blanks to the length of the longer operand for the comparison.
- If no other relationship can be determined after the strings are exhausted, the strings are equal.

The collating sequence depends partially on the processor; however, equality tests **.EQ.** and **.NE.** do not depend on the processor collating sequence and can be used on any processor.

Logical Expressions

A logical expression specifies a logical computation that yields a logical value. The simplest form of a logical expression is one of the following:

- logical constant
- logical variable reference
- logical array element reference
- logical function reference
- relational expression

Construct complicated logical expressions from one or more logical operands together with logical operators and parentheses.

Logical Operators

Table 3−5defines the Fortran logical operators.

Table 3−5 Logical Operators

Logical Operator	Meaning
.NOT.	Logical negation
.AND.	Logical conjunt
.OR.	Logical disjunct
.EQV.	Logical equivalence
.NEQV.	Logical exclusive or
.XOR.	Same as .NEQV.

All logical operators require at least two operands, except the logical negation operator**.NOT.** , which requires only one.

A logical expression containing two or more logical operators is evaluated based on a precedence relation between the logical operators. This precedence, from highest to lowest, is

- **.NOT.**
- **.AND.**
- **.OR.**
- **.EQV.** and **.NEQV.**
- **.XOR.**

For example, in the following expression

```
W .NEQV. X .OR. Y .AND. Z
```

the operators are executed in the following sequence:

1. **Y .AND. Z** is evaluated first (**A** represents the result).
2. **X .OR. A** is evaluated second (**B** represents the result).
3. **W .NEQV. B** is evaluated to produce the final result.

You can use parentheses to override the precedence of the operators.

Logical Operands

Logical operands specify values with a logical data type. The forms of a logical operands are

- logical primary
- logical factor
- logical term
- logical disjunct
- logical expression

Logical Primary

The logical *primary* is the basic component of a logical expression. The forms of a logical primary are

- logical constant
- symbolic name of a logical constant
- integer or logical variable reference
- logical array element reference
- integer or logical function reference
- relational expression
- integer or logical expression in parentheses

When an integer appears as an operand to a logical operator, the other operand is promoted to type integer if necessary and the operation is performed on a bit−by−bit basis producing an integer result. Whenever an arithmetic datum appears in a logical expression, the result of that expression will be of type integer because of type promotion rules. If necessary, the result can be converted back to **LOGICAL**.

Do not specify two logical operators consecutively and do not use implied logical operators.

Logical Factor

The *logical factor* uses the logical negation operator **.NOT.** to reverse the logical value to which it is applied. For example, applying **.NOT.** to a false relational expression makes the expression true. Therefore, if **UP** is true, **.NOT. UP** is false. The logical factor has the following forms:

- logical primary
- **.NOT.** logical primary

Logical Term

The *logical term* uses the logical conjunct operator **.AND.** to combine logical factors. It takes the forms

- Logical factor
- Logical term **.AND.** logical factor

In evaluating a logical term with two or more **.AND.** operators, the logical factors are combined from left to right. For example, **X .AND. Y .AND. Z** has the same interpretation as **(X .AND. Y) .AND. Z**.

Logical Disjunct

The *logical disjunct* is a sequence of logical terms separated by the **.OR.** operator and has the following two forms:

- Logical term

- Logical disjunct **.OR.** logical term

In an expression containing two or more **.OR.** operators, the logical terms are combined from left to right in succession. For example, the expression **X .OR. Y .OR. Z** has the same interpretation as **(X .OR. Y) .OR. Z**.

Logical Expression

At the highest level of complexity is the *logical expression.* A logical expression is a sequence of logical disjuncts separated by the **.EQV**., **.NEQV.**, or **.XOR.** operators. Its forms are

- logical disjunct
- logical expression **.EQV.** logical disjunct
- logical expression **.NEQV.** logical disjunct
- logical expression **.XOR.** logical disjunct

The logical disjuncts are combined from left to right when a logical expression contains two or more **.EQV.**, **.NEVQ.**, or **.XOR.** operators.

A logical constant expression is a logical expression in which each primary is a logical constant, the symbolic name of a logical constant, a relational expression in which each primary is a constant, or a logical constant expression enclosed in parentheses. A logical constant expression can contain arithmetic and character constant expressions but not variables, array elements, or function references.

Interpretation of Logical Expressions

In general, logical expressions containing two or more logical operators are executed according to the hierarchy of operators described previously, unless the order has been overridden by the use of parentheses. Table 3−6defines the form and interpretation of the logical expressions.

Table 3−6 Logical Expressions

IFA=	B=	THEN .NOT.B	A.AND.B	A.OR.B	A.EQV.B	A.XOR.B A.NEQV.B
F	F	T	F	F	T	F
F	T	F	F	T	F	T
T	F	-	F	T	F	T
T	T	-	T	T	T	F

Evaluating Expressions in General

Several rules are applied to the general evaluation of Fortran expressions. This section covers the priority of the different Fortran operators, the use of parentheses in specifying the order of evaluation, and the rules for combining operators with operands.

Note: Any variable, array element, function, or character substring in an expression must be defined with a value of the correct type at the time it is referenced.

Precedence of Operators

Certain Fortran operators have precedence over others when combined in an expression. The previous sections have listed the precedence among the arithmetic, logical, and expression operators. No precedence exists between the relational operators and the single character operator (//). On the highest level, the precedence among the types of expression operators, from highest to lowest, is

- arithmetic
- character
- relational
- logical

Integrity of Parentheses and Interpretation Rules

Use parentheses to specify the order in which operators are evaluated within an expression. Expressions within parentheses are treated as an entity.

In an expression containing more than one operation, the processor first evaluates any expressions within parentheses. Subexpressions within parentheses are evaluated beginning with the innermost subexpression and proceeding sequentially to the outermost. The processor then scans the expression from left to right and performs the operations according to the operator precedence described previously.

Chapter 4

Specification Statements

This chapter contains the following subsections:

- "AUTOMATIC, STATIC"
- "BLOCK DATA"
- "COMMON"
- "DATA"
- "Data Type Statements"
- "DIMENSION"
- "EQUIVALENCE"
- "EXTERNAL"
- "IMPLICIT"
- "INTRINSIC"
- "NAMELIST"
- "PARAMETER"
- "POINTER"
- "PROGRAM"
- "RECORD"
- "SAVE"
- "STRUCTURE / UNION"
- "VOLATILE"

Specification statements are non−executable Fortran statements that provide the processor with information about the nature of specific data and the allocation of storage space for this data.

The specification statements are summarized below.

AUTOMATIC, **STATIC**
: Controls the allocation of storage to variables and the initial value of variables within called subprograms.

BLOCK DATA
: First statement in a block data subprogram used to assign initial values to variables and array elements in named common blocks.

COMMON
: Declares variables and arrays to be put in a storage area that is accessible to multiple program units, thus allowing program units to share data without using arguments.

DATA
: Supplies initial values of variables, array elements, arrays, or substrings.

Data type	Explicitly defines the type of a constant, variable, array, external function, statement function, or dummy procedure name. Also, can specify dimensions of arrays and the length of the character data.
DIMENSION	Specifies the symbolic names and dimension specifications of arrays.
EQUIVALENCE	Specifies the sharing of storage units by two or more entities in a program unit, thus associating those entities.
EXTERNAL	Identifies external or dummy procedure.
IMPLICIT	Changes or defines default implicit type of names.
INTRINSIC	Identifies intrinsic function or system subroutine.
NAMELIST	Associates a group of variables or array names with a unique group name.
PARAMETER	Gives a constant a symbolic name.
POINTER	Establishes pairs of variables and pointers.
PROGRAM	Defines a symbolic name for the main program.
RECORD	Creates a record in the format specified by a previously declared **STRUCTURE** statement.
SAVE	Retains the values of variables and arrays after execution of a **RETURN** or **END** statement in a subprogram.
STRUCTURE	Defines a record structure that can be referenced by one or more **RECORD** statement.
VOLATILE	Prevents the compiler from optimizing specified variables, arrays, and common blocks of data.

Detailed descriptions of these statements follow in alphabetical order.

AUTOMATIC, STATIC

STATIC and **AUTOMATIC** statements control, within a called subprogram, the allocation of storage to variables and the initial value of variables.

Syntax

```
{STATIC | AUTOMATIC} v [,v] ...
```

where *v* is the name of a previously declared variable, array, array declarator, symbolic constant, function, or dummy procedure.

Method of Operation

Table 4−1summarizes the differences between *static* and *automatic* variables on entry and exit from a subprogram.

Table 4–1 Static and Automatic Variables

	AUTOMATIC	STATIC
Entry	Variables are unassigned. They do not reflect any changes caused by the previous execution of the subprogram.	Values of the variables in the subprogram are unchanged since the last execution of the subprogram.
Exit	The storage area associated with the variable is deleted.	The current value of the variable is retained in the static storage area.

AUTOMATIC variables have two advantages:

- The program executes more efficiently by taking less space and reducing execution time.
- They permit recursion; a subprogram can call itself either directly or indirectly, and the expected values are available on either a subsequent call or a return to the subprogram.

Rules for Use

- By default, unless you specify the **-static** command line option (described in the *f77*(1) manual page and Chapter 1 of the *Fortran 77 Programmer's Guide*), all variables are **AUTOMATIC** except
  - initialized variables
  - common blocks
  - variables used in **EQUIVALENCE** statements
- Override the command line option in effect for specific variables by specifying as applicable the **AUTOMATIC** or **STATIC** keywords in the variable type statements, as well as in the **IMPLICIT** statement.
- Any variable in **EQUIVALENCE**, **DATA**, or **SAVE** statements is static regardless of any previous **AUTOMATIC** specification.

Example

```
REAL   length, anet, total(50)
STATIC length, anet, total
COMPLEX i, B(20), J(2,3,5)
STATIC i
IMPLICIT INTEGER(f,m-p)
IMPLICIT STATIC (f,m-p)
```

BLOCK DATA

BLOCK DATA is the first statement in a block data subprogram. It assigns initial values to variables and array elements in named common blocks.

Syntax

```
BLOCK DATA [sub]
```

where *sub* is the symbolic name of the block data subprogram in which the **BLOCK DATA** statement appears.

Method of Operation

A block data subprogram is a non–executable program unit with a **DATA** statement as its first statement, followed by a body of specification statements and terminated by an **END** statement. The types of specification statements include **COMMON**, **DATA**, **DIMENSION**, **EQUIVALENCE**, **IMPLICIT**, **PARAMETER**, **SAVE**, **STRUCTURE** declarations, and type statements. A block data subprogram can also contain comment lines.

Only entities in named common blocks or entities associated with an entity in a common block can be initially defined in a block data subprogram.

Rules for Use

- The optional name *sub* is a global name and must be unique. Thus, **BLOCK DATA** subprograms cannot have the same external name.
- An executable program can contain more than one block data subprogram but cannot contain more than one unnamed block data subprogram.
- A single block data subprogram can initialize the entities of more than one named common block.

COMMON

The **COMMON** statement declares variables and arrays so that they are put in a storage area that is accessible to multiple program units, thus allowing program units to share data without using arguments.

Syntax

```
COMMON [/[cb]/] nlist [[,]/[cb]/ nlist]...
```

where

cb	is a common block name.
nlist	is a list of variable names, array names, array declarators, or records.

Method of Operation

A storage sequence, composed of a series of storage units that are shared between program units, is referred to as *common storage*. For each common block, a common block storage sequence is formed consisting of the storage sequences of all entities in the list of variables and arrays for that common block. The order of the storage sequence is the same as its order of appearance in the list. In each **COMMON** statement, the entities specified in the common block list *nlist* following a block name *cb* are declared to be in common block *cb*.

In an executable program, all common blocks with the same name have the same first storage unit. This establishes the association of data values between program units.

The storage sequence formed above is extended to include all storage units of any storage sequence associated with it by equivalence association.

Fortran has the following types of common storage:

- *Blank* common storage does not have an identifying name and can be accessed by all program units in which it is declared. One blank common area exists for the complete executable program.
- *Named* common storage has an identifying name and is accessible by all program units in which common storage with the same name is declared.

You can initially define entities in a named common block by using the **DATA** initialization statement in a **BLOCK DATA** subprogram. However, you *cannot* use the **DATA** statement to initialize entities in blank common block.

The number of storage units needed to store a common block is referred to as its size. This number includes any extensions of the sequence resulting from equivalence association. The size of a named common block must be the same in *all* program units in which it is declared. The size of blank common block need *not* be the same size in all program units.

Rules for Use

- A variable name, array name, array declarator, or record can appear only once in all common block lists within a program unit.
- Specify a blank common block by omitting the common block name *cb* for each list. Thus, omitting the first common block name places entities appearing in the first *nlist* in a blank common block.
- Omitting the first *cb* makes the first two slashes optional. Two slashes without a block name between them declare the entities in the following list to be in a blank common block.
- Any common block name *cb* or an omitted *cb* for a blank common block can occur more than once in one or more **COMMON** statements in a program unit. The list following each appearance of the same common block name is treated as a continuation of the list for that common block name.
- As an extension to the standard, a named common block can be declared as having different sizes in different program units. If the common block is not initially defined with a **DATA** statement, its size will be that of the longest common block declared. However, if it is defined in one of the program units with **DATA** statements, then its size is the size of the defined common block. In other words, to work correctly, the named common block must be declared with the longest size when it is defined, even though it can be declared with shorter sizes somewhere else. Defining a common block multiple times produces incorrect results.
- The compiler aligns entities in a common block on 32−bit boundaries. You can change this alignment using the compiler switches **-align8** and **-align16**. However, these changes can degrade

performance. See the *Fortran 77 Programmer's Guide* for more information.

Restrictions

- Names of dummy arguments of an external procedure in a subprogram must not appear in a common block list.
- A variable name that is also a function name must not appear in the list.

Examples

The following equivalent statements define a blank common block. (Note that these two **COMMON** statements cannot appear in the same program unit).

```
COMMON //F,X,B(5)
COMMON F,X,B(5)
```

This declaration

```
COMMON /LABEL/NAME,AGE,DRUG,DOSE//Y(33),
 Z,/RECORD/,DOC, 4 TIME(5), TYPE(8)
```

makes the following **COMMON** storage assignments:

- **NAME**, **AGE**, **DRUG**, and **DOSE** are placed in common block **LABEL**.
- **Y** and **Z** are placed in a blank common block.
- **DOC**, **TIME**, and **TYPE** are placed in a common block **RECORD**.

The following program contains two **COMMON** statements: one in the calling program and one in the subroutine. Both define the same four entities in the **COMMON** even though each common statement uses a unique set of names. The calling program can access **COMMON** storage through the entities **TOT**, **A**, **K**, and **XMEAN**. Subroutine **ADD** has access to the same common storage through the use of the entities **PLUS**, **SUM**, **M**, and **AVG**.

```
c  THIS PROGRAM READS VALUES AND PRINTS THEM
c  SUM AND AVERAGE
   COMMON TOT, A(20), K, XMEAN
   READ (5,10) K, ( A(I), I = 1, K)
   CALL ADD
   WRITE (6,20) TOT, XMEAN
10 FORMAT (I5/F(10.0))
20 FORMAT (5X,5HSUM =,2X,F10.4/5X,
  +       12HMEAN VALUE =,2X,F10.4)
   STOP
c
c  THIS SUBROUTINE CALCULATES THE SUM AND AVERAGE
c
```

```
   COMMON PLUS, SUM(20), M, AVG
   PLUS = SUM (1)
   DO 5 I = 2, M
5  PLUS = SUM (I) + PLUS
   AVG = PLUS / FLOAT (M)
   END
```

DATA

The **DATA** statement supplies initial values of variables, array elements, arrays, or substrings.

Syntax

`DATA` *nlist*/*clist*/ `[[ , ]` *nlist*/*clist*/`]` …

where

nlist is a list of variable names, array names, array element names, substring names or implied-**DO** lists (described later in this chapter) separated by commas.

clist *clist* is composed of one or more elements, separated by commas, of either of the following forms:*cr*c* where *c* is a constant or the symbolic name of a constant. *r* is a nonzero, unsigned integer constant or the symbolic name of a positive integer constant. The second form implies *r* successive appearances of the constant c.

Method of Operation

In data initialization, the first value in *clist* is assigned to the first entity in *nlist*, the second value in *clist* to the second entity in *nlist*, and so on. There is a one−to−one correspondence between the items specified by *nlist* and the constants supplied in *clist.* Hence, each *nlist* and its corresponding *clist* must contain the same number of items and must agree in data type. If necessary, the *clist* constant is converted to the type or length of the *nlist* entity exactly as for assignment statements.

If the length of the character entity in *nlist* is greater than the length of its corresponding character constant in *clist*, then blank characters are added to the right of the character constant. But if the length of the character entity in *nlist* is less than that of its corresponding constant in *clist*, the extra right most characters in the constant are ignored; only the left most characters are stored. Each character constant initializes only one variable, array element, or substring.

As an enhancement to Fortran 77, you can define an arithmetic or logical entity initially using a Hollerith constant for *c* in a *clist*, using the form

n**H***x1 x2 x3 … xn*

where

n is the number of characters *xn.*

xi is the actual characters of the entity.

The value of *n* must be > *g*, where *g* is the number of character storage units for the corresponding entity.

If $n < g$, the entity is initially defined with the n Hollerith characters extended on the right with $g - n$ blank characters. The compiler generates a warning message for data initializations of this type.

Rules for Use

- Each *nlist* and its corresponding *clist* must have the same number of items and must correspond in type when either is **LOGICAL** or **CHARACTER**. If either is of arithmetic type, then the other must be of arithmetic type.
- If an unsubscripted array name is specified in *nlist*, the corresponding *clist* must contain one constant for each element of the array.
- If two entities are associated in common storage, only one can be initialized in a **DATA** statement.
- Each subscript expression in *nlist* must be an integer constant expression, except for implied-**DO** variables.
- Each substring expression in *nlist* must be an integer constant expression.
- A numeric value can be used to initialize a character variable or element. The length of that character variable or array element must be one, and the value of the numeric initializer must be in the range 0 through 255.
- An untyped hexadecimal, octal, or binary constant can be used to initialize a variable or array element. If the number of bits defined by the constant is less than the storage allocation for that variable or array element, then leading zeros are assumed. If the number of bits exceed the number of bits of storage available for the variable or array element, then the leading bits of the constant are truncated accordingly.
- A Hollerith constant can be used to initialize a numeric variable or array element. The rules for Hollerith assignment apply.

Restrictions

- The list *nlist* cannot contain names of dummy arguments, functions, and entities in blank common, or those associated with entities in blank common.
- Do not initialize a variable, array element, or substring more than once in an executable program. If you do, the subsequent initializations will override the previous ones.
- If a common block is initialized by a **DATA** statement in a program unit, it cannot be initialized in other program units.

Example

Given the following declarations,

```
REAL          A (4), b
LOGICAL       T
COMPLEX       C
```

```
      INTEGER        P, K(3), R
      CHARACTER*5    TEST(4)
      PARAMETER      (P=3)
      DATA   A,B/0.,12,5.12E5,0.,6/, T/.TRUE./,
     +              C/(7.2, 1.234)/,K/P*0/,
     +              TEST/3*'MAYBE','DONE?'/
```

the **DATA** statement above defines the variables declared immediately preceding it as follows:

```
A(1) = .0E+00    A(2) = .12E+02
A(3) = .512E+06  A(4) = .0E+00
B = 6
T = .TRUE.
C = (.72E+01, .1234+01)
K(1) = 0   K(2) = 0 K(3) = 0
TEST(1) = 'MAYBE' TEST(2) = 'MAYBE'
TEST(3) = 'MAYBE' TEST(4) = 'DONE?'
```

The following statements are examples of implied-**DO** statements using **DATA** statements:

```
DATA LIMIT /1000/, (A(I), I= 1,25)/25*0/
DATA ((A(I,J), J = 1,5), I = 1,10)/50*1.1/
DATA (X(I,I), I = 1,100) /100 * 1.1/
DATA ((A(I,J), J = 1,I), I =1,3)/11,21,22,31,32,33/
```

Data Type Statements

The data type statement explicitly defines the type of a constant, variable, array, external function, statement function, or dummy procedure name. It can also specify dimensions of arrays and the length of character data. The two kinds of data type statements are numeric and character.

Numeric Data Types

Use numeric data types to

- override implicit typing
- explicitly define the type of a constant, variable, array, external function, statement function, or dummy procedure name
- specify dimensions of arrays

Syntax

type v [**len*] [*/clist/*] [, *v*[**len*]*/clist/*]]

where

type is one of the keywords listed in Table 4–2 *v* is a variable name, array name, array declarator, symbolic name of a constant, function name, or dummy procedure name.

len is one of the acceptable lengths for the data type being declared; *len* is one of the following: an unsigned, nonzero integer constant; a positive−value integer constant expression enclosed in parentheses; or an asterisk enclosed in parentheses (*). If the type being declared is an array, *len* follows immediately after the array name.

clist is a list of values bounded by slashes; the value becomes the initial value of the type being declared.

Table 4−2 Keywords for Type Statements

INTEGER	COMPLEX
INTEGER*1	DOUBLE COMPLEX
BYTE	COMPLEX*8
INTEGER*2	COMPLEX*16
INTEGER*4	
LOGICAL	REAL
LOGICAL*1	REAL*4
LOGICAL*2	REAL*8
LOGICAL*4	REAL*16
DOUBLE PRECISION	

Note: When the complier encounters a **REAL*16** declaration, it issues a warning message. **REAL*16** items are allocated 16 bytes of storage per element, but only the first 8 bytes of each element are used; those 8 bytes are interpreted according to the format for **REAL*8** floating numbers.

The following pairs of keywords are synonymous:

- **BYTE** and **INTEGER*1**
- **REAL** and **REAL*4**
- **DOUBLE PRECISION** and **REAL*8**
- **COMPLEX** and **COMPLEX*8**
- **DOUBLE COMPLEX** and **COMPLEX*16**
- **LOGICAL** and **LOGICAL*4**

See Chapter 2 of the *Fortran 77 Programmer's Guide* for information on the alignment, size, and value ranges of these data types.

Method of Operation

The symbolic name of an entity in a type statement establishes the data type for that name for all its subsequent appearances in the program unit in which it is declared.

The *type* specifies the data type of the corresponding entities. That is, the **INTEGER** statement explicitly

The *type* specifies the data type of the corresponding entities. That is, the **INTEGER** statement explicitly declares entities of type integer and overrides implicit typing of the listed names. The **REAL** statement specifies real entities, the **COMPLEX** statement specifies complex entities, and so on.

Rules for Use

- Type statements are optional and must appear in the beginning of a program unit. However, type statements can be preceded by an **IMPLICIT** statement.
- Symbolic names, including those declared in type statements, have the scope of the program unit in which they are included.
- A program unit can contain type statements that begin with identical keywords.
- Do not explicitly specify the type of a symbolic name more than once within a program unit.
- Do not use the name of a main program, subroutine, or block data subprogram in a type statement.
- The compiler provides a **DOUBLE COMPLEX** version of the functions in Table 4−3

Table 4−3 Double Complex Functions

Name	Purpose
dcmplx	Explicit type conversion
dconjg	Complex conjugate
dimag	Imaginary part of complex argument
zabs	Complex absolute value

- The **-i2** compiler option (see the *f77*(1) man page or Chapter 2 of the *Fortran 77 Programmer's Guide*) causes the following to take place:
  - converts integer constants whose values are within the range allowed for the **INTEGER*2** data types to **INTEGER*2**
  - converts the data type of variable returned by a function to **INTEGER*2**, where possible
  - ensures that variables of type **LOGICAL** occupy the same amount of storage as **INTEGER*2** variables

Examples

```
REAL length, anet, TOTAL(50)
INTEGER hour, sum(5:15), first, uvr(4,8,3)
LOGICAL bx(1:15,10), flag, stat
COMPLEX I, B(20), J(2,3,5)
```

The code above declares that

- **length** and **anet** are names of type real. The specification of **anet** confirms implicit typing using the first letter of the name and could have been omitted in the **REAL** statement.
- **TOTAL** is a real array.
- **hour** and **first** are integer names. **uvr** and **sum** are integer arrays and illustrate the use of the type

hour and **first** are integer names. **uvr** and **sum** are integer arrays and illustrate the use of the type statement to specify the dimensions of an array. Note that when an array is dimensioned in a type statement, a separate **DIMENSION** statement to declare the array is not permitted.

- **flag** and **stat** are logical variables; **bx** is a logical array.
- **I** is a complex variable; **B** and **J** are complex arrays.

Character Data Types

Character data type statements declare the symbolic name of a constant, variable, array, external function, statement function, or dummy procedure name and specify the length of the character data.

Syntax

`CHARACTER [*`*len*` [,]] `*nam*` [,`*nam*`] . . .`

where

len — is a length specification that gives the length, in number of characters, of a character variable, character array element, character constant, or character function. *len* is one of the following:

- an unsigned, nonzero integer constant
- a positive–value integer constant expression enclosed in parentheses
- an asterisk enclosed in parentheses (*)

nam — is one of the following: *v* [**len*] where *v* is a variable name, symbolic name of a constant, function name, or dummy procedure name*a* [(*d*)] [**len*] where *a*(*d*) is an array declarator

Rules for Use

- The length specification *len* that follows the keyword **CHARACTER** denotes the length of each entity in the statement that does not have its own length specification.
- A length specification immediately following an entity applies only to that entity. The length specified when an array is declared applies to each array element.
- If no length specification is given, a length of one is assumed.
- The length specifier of (*) can be used only for names of external functions, dummy arguments of an external procedure, and character constants.
  - For a character constant, the (*) denotes that the length of the constant is determined by the length of the character expression given in the **PARAMETER** statement.
  - For a dummy argument of an external procedure, the (*) denotes that the length of the dummy argument is the length of the actual argument when the procedure is invoked. If the associated actual argument is an array name, the length of the dummy argument is the length of an element of the actual array.

  - For an external function name, the (*) denotes that the length of the function result value and the local variable with the same name as the function entry name is the length that is specified in the program unit in which it is referenced. Note that the function name must be the name of an entry to the function subprogram containing this **TYPE** statement.

- If an actual *len* is declared for an external function in the referencing program unit and in the function definition, *len* must agree with the length specified in the subprogram that specifies the function. If not, then the function definition must use the asterisk (*) as covered previously, but the actual *len* in the referencing unit must not be (*).

- The length specified for a character statement function or statement function dummy argument of type character must be an integer constant expression.

Example

```
CHARACTER name*40, gender*1, pay(12)*10
```

The above declaration defines

- **name** as a character variable with a length of 40
- **gender** as a character variable with a length of one
- **pay** as a character array with 12 elements, each of which is 10 characters in length

DIMENSION

The **DIMENSION** statement specifies the symbolic names and dimension specifications of arrays.

Syntax

DIMENSION *a*(*d*) [,*a*(*d*)] ...

where *a*(*d*) is an array declarator.

To be compatible with PDP–11 Fortran, the **VIRTUAL** statement is synonymous with the **DIMENSION** statement and carries the identical meaning.

Method of Operation

A symbolic name *x* appears in a **DIMENSION** statement causing an array *x* to be declared in that program unit.

Rules for Use

- The dimension specification of an array can appear only once in a program unit.
- The name of an array declared in a **DIMENSION** statement can appear in a type statement or a **COMMON** statement without dimensioning information.

Examples

The following **DIMENSION** statement declares **z** as an array of 25 elements, **a** as an array of 36 elements (6 x 6), and **ams** as an array of 50 elements (2 x 5 x 5).

```
DIMENSION z(25), a(6,6), ams(2,5,5)
```

EQUIVALENCE

The **EQUIVALENCE** statement allows two or more entities in a program unit to share storage units, thus associating those entities. This statement allows the same information to be referenced by different names in the same program unit.

Syntax

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

where *nlist* is a list of variable names, array element names, array names, and character substring names.

Method of Operation

The storage sequences of the entities in the list must have the same first storage unit. This requirement associates the entities in the list or other elements as well. The **EQUIVALENCE** statement only associates storage units and does not cause type conversion or imply mathematical equivalence. Thus, if a variable and an array are equivalenced, the variable does not assume array properties and vice versa.

Character entities can be associated by equivalence only with other character entities. Specify the character entities, character variables, character array names, character array element names, or character substring names. Association is made between the first storage units occupied by the entities appearing in the equivalence list of an **EQUIVALENCE** statement. This statement can associate entities of other character elements as well. The lengths of the equivalenced character entities are not required to be equal.

Variables and arrays can be associated with entities in common storage to lengthen the common block. However, association through the use of the **EQUIVALENCE** statement must not cause common storage to be lengthened by adding storage units before the first storage unit in the common block.

Rules for Use

- Each subscript expression or substring expression in an equivalence list must be an integer constant expression.
- If an array element name is specified in an **EQUIVALENCE** statement, the number of subscript expressions must be the same as the number of dimensions declared for that array.
- An array name without a subscript is treated as an array element name that identifies the first element of the array.
- Multidimensional array elements can be referred to in an **EQUIVALENCE** statement with only one subscript. The compiler considers the array to be one–dimensional according to the array element ordering of Fortran. Consider the following example:

```
DIMENSION a(2,3), b(4:5,2:4)
```

The following shows a valid **EQUIVALENCE** statement using the arrays **a** and **b**:

```
EQUIVALENCE (a(1,1), b(4,2))
```

The following example achieves the same effect:

```
EQUIVALENCE (a(1), b(4))
```

The lower−bound values in the array declaration are always assumed for missing subscripts (in the above example, 1 through 3 for array **a** and 2 through 4 for array **b**).

Restrictions

- Names of dummy arguments of an external procedure in a subprogram cannot appear in an equivalence list.
- A variable name that is also a function name cannot appear in the list.
- A storage unit can appear in no more than one **EQUIVALENCE** storage sequence.
- An **EQUIVALENCE** statement cannot specify non−consecutive storage positions for consecutive storage units.
- An **EQUIVALENCE** statement cannot associate a storage unit in one common block with any storage unit in a *different* common block.

Example 1

The two statements below are represented in storage as shown in Figure 4−1

```
DIMENSION M(3,2),P(6)
EQUIVALENCE (M(2,1),P(1))
```

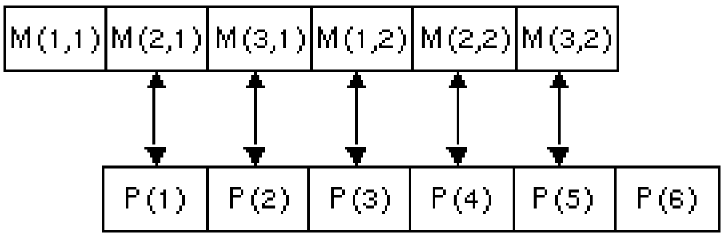

Figure 4−1 Storage Representation of an EQUIVALENCE Statement

Example 2

The two statements below cause the logical representation in storage shown in Figure 4−2

```
CHARACTER ABT*6, BYT(2)*4, CDT*3
EQUIVALENCE (ABT, BYT(1)),(CDT, BYT(2))
```

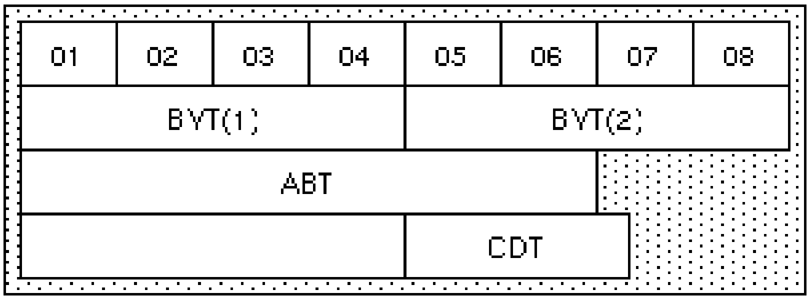

Figure 4–2 Logical Representation of an EQUIVALENCE Statement

Example 3

The following statements are invalid because they specify non–consecutive storage positions for consecutive storage units.

```
REAL A(2)
DOUBLE PRECISION S(2)
EQUIVALENCE (A(1), S(1)), (A(2), S(2))
```

Note that a double–precision variable occupies two consecutive numeric storage units in a storage sequence.

EXTERNAL

The **EXTERNAL** statement specifies a symbolic name to represent an external procedure or a dummy procedure. The symbolic name can then be used as an actual argument in a program unit.

Syntax

`EXTERNAL` *proc* `[,`*proc*`] ...`

where *proc* is the name of an external procedure or dummy procedure.

Rules for Use

- An external procedure name or a dummy procedure name must appear in an **EXTERNAL** statement in the program unit if the name is to be used as an actual argument in that program unit.
- If an intrinsic function name appears in an **EXTERNAL** statement, indicating the existence of an external procedure having that name, the intrinsic function is not available for use in the same program unit in which the **EXTERNAL** statement appears.
- A symbolic name can appear only once in all the **EXTERNAL** statements of a program unit.

- A **NOF77** qualifier in an **OPTIONS** statement or the **-nof77** command line option permits the following:
  - Intrinsic function names can appear in the list of subprograms in an **EXTERNAL** statement.
  - An asterisk (*) can precede program names listed in an **EXTERNAL** statement. This indicates that a user–supplied function has the same name as a Fortran intrinsic function and that the user–supplied function is to be invoked.

Restriction

Do not specify a statement function name in an **EXTERNAL** statement.

Example

Consider the following statements:

```
EXTERNAL G
CALL SUB1 (X,Y,G)
```

and the corresponding subprogram:

```
SUBROUTINE SUB1 (RES, ARG, F)
RES = F(ARG)
END
```

The dummy argument **F** in subroutine **SUB1** is the name of another subprogram; in this case, the external function **G**.

IMPLICIT

The **IMPLICIT** statement changes or defines default–implicit types of names. This section explains the three syntactic forms of the **IMPLICIT** statement.

Syntax 1

`IMPLICIT` *typ* `(`*a*`[,`*a*`]...) [,`*typ*`(`*a*`[,`*a*`]...)]...`

where

typ	is a valid data type.
a	is either a single *alphabetic* character or a range of letters in alphabetical order. A range of letters is specified as *l*1 - *l*2, where *l*1 and *l*2 are the first and last letters of the range, respectively.

An **IMPLICIT** statement specifies a type for all variables, arrays, external functions, and statement functions for which no type is explicitly specified by a type statement. If a name has not appeared in a type statement, then its type is implicitly determined by the first character of its name. The **IMPLICIT** statement establishes which data type (and length) will be used for the indicated characters.

By default, names beginning with the alphabetic characters A through H or O through Z are implicitly typed **REAL**; names beginning with I, J, K, L, M, or N are implicitly typed **INTEGER**. Use the **IMPLICIT** statement to change the type associated with any individual letter or range of letters.

An **IMPLICIT** statement applies only to the program unit that contains it and is overridden by a type statement or a **FUNCTION** statement in the same subprogram.

Syntax 2

```
IMPLICIT {AUTOMATIC | STATIC} (a[,a]...)
  [,typ (a[,a]...)]
```

An **AUTOMATIC** or **STATIC** keyword in an **IMPLICIT** statement causes all associated variables to be assigned automatic or static storage characteristics. See the description of the **AUTOMATIC** and **STATIC** statements earlier in this chapter for information on their function. An example using these keywords is also given.

Syntax 3

```
IMPLICIT {UNDEFINED | NONE}
```

Note: UNDEFINED and **NONE** are synonymous and, therefore, perform the same function.

When a type is not declared explicitly for a variable, the implicit data typing rules cause a default type of **INTEGER** to apply if the first letter of the variable is i, j, k, l, m, or n or **REAL** if the first letter is any other alphabetic character.

Use the **IMPLICIT UNDEFINED** statement, **IMPLICIT NONE** statement, or the **-u** command line option to turn off the implicit data typing.

Using Syntax 3 of the **IMPLICIT** statement within a program allows you to override the default assignments given to individual characters; the **-u** command line option (see Chapter 1 of the *Fortran 77 Programmer's Guide*) overrides the default assignments for all alphabetic characters.

The following declaration

```
IMPLICIT UNDEFINED
```

turns off the implicit data typing rules for all variables. The example has the same effect as specifying the **-u** command line option.

Rules for Use

The following rules are for all three syntactic forms of the **IMPLICIT** statement.

- **IMPLICIT** statements must precede all other specification statements except **PARAMETER** statements.
- Multiple **IMPLICIT** statements are allowed in a program unit.
- **IMPLICIT** statements cannot be used to change the type of a letter more than once inside a program unit. Because letters can be part of a range of letters as well as stand alone, ranges of letters cannot overlap.

- Lowercase and uppercase alphabetic characters are *not* distinguished. Implicit type is established for *both* the lower– and uppercase alphabetic characters or range of alphabetic characters regardless of the case of *l*1 and *l*2.
- The **-u** command line option turns off all default data typing and any data typing explicitly specified by an **IMPLICIT** statement.

Examples

Consider the following example:

```
IMPLICIT NONE
IMPLICIT INTEGER (F,M-P)
IMPLICIT STATIC (F,M-P)
IMPLICIT REAL (B,D)
INTEGER bin, dale
```

The previous statements declare that

- All variables with names beginning with the letters F(f), M(m), N(n), O(o), or P(p) are of type **INTEGER** and are assigned the **STATIC** attribute.
- All variables with names beginning with the letter B(b) or D(d) are of type **REAL**, except for variables **bin** and **dale**, which are explicitly defined as type **INTEGER**.

The following four **IMPLICIT** statements are equivalent:

```
IMPLICIT CHARACTER (g - k)
IMPLICIT CHARACTER (g - K)
IMPLICIT CHARACTER (G - k)
IMPLICIT CHARACTER (G - K)
```

INTRINSIC

INTRINSIC statements associate symbolic names with intrinsic functions and system subroutines. The name of an intrinsic function can be used as an actual argument.

Syntax

INTRINSIC *func* [,*func*] ...

where *func* is a name of intrinsic functions.

Rules for Use

- The name of every intrinsic function or system subroutine used as an actual argument must appear in an **INTRINSIC** statement in that program unit.
- A symbolic name can appear only once in all of the **INTRINSIC** statements of a program unit.

Restrictions

- The same name cannot appear in both an **INTRINSIC** and an **EXTERNAL** statement in the same program unit.
- The same name can appear only once in all the **INTRINSIC** statements of a program unit.
- The names of intrinsic functions that perform type conversion, test lexical relationship, or choose smallest/largest value cannot be passed as actual arguments. These functions include the conversion, maximum−value, and minimum−value functions listed in Appendix A, "Intrinsic Functions."

Examples

Consider the following statements:

```
INTRINSIC ABS
CALL ORD (ABS, ASQ, BSQ)
```

and the corresponding subprogram:

```
SUBROUTINE ORD(FN,A,B)
A = FN (B)
RETURN
END
```

In the above example, the **INTRINSIC** statement allows the name of the intrinsic function **ABS** (for obtaining the absolute value) to be passed to subprogram **ORD**.

NAMELIST

The **NAMELIST** statement associates a group of variables or array names with a unique group−name in a namelist−directed I/O statement.

Syntax

`NAMELIST /`*group−name*`/`*namelist*`[,] /`*group−name*`/ `*namelist*`...`

where *group−name* is the name to be associated with the variables or array names defined in *namelist*. Each item in *namelist* must be separated by a comma.

Rules for Use

- The items in *namelist* are read or written in the order they are specified in the list.
- The items can be of any data type, which can be specified either explicitly or implicitly.
- The following items are not permitted in *namelist*:
  - dummy arguments
  - array elements

- character substrings
- records
- record fields

See also the description of the **READ** and **WRITE** statements in Chapter 8, "Input/Output Statements," for more information on namelist−directed I/O.

Examples

In the following statement, **input**, when specified to a namelist−directed I/O statement, refers to **item** and **quantity**; likewise, **output** refers to **item** and **total**:

```
NAMELIST /input/ item, quantity /output/ item, total
```

PARAMETER

The **PARAMETER** statement assigns a symbolic name to a constant.

Syntax

Format 1

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

Format 2

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

where

p	is a symbolic name.
e	is a constant, constant expression, or the symbolic name of a constant.

Method of Operation

The value of the constant expression e is given the symbolic name *p*. The statement defines *p* as the symbolic name of the constant. The value of the constant is the value of the expression *e* after conversion to the type name *p*. The conversion, if any, follows the rules for assignment statements.

Format 1, which has bounding parentheses, causes the symbolic name to be typed either of the following ways:

- According to a previous explicit type statement.
- If no explicit type statement exists, the name is typed according to its initial letter and the implicit rules in effect. See the description of the **IMPLICIT** statement in "IMPLICIT" for details.

Format 2, which has no bounding parentheses, causes the symbolic name to be typed by the form of the actual constant that it represents. The initial letter of the name and the implicit rules do not affect the data type.

A symbolic name in a **PARAMETER** statement has the scope of the program unit in which it was declared.

Rules for Use

- If *p* is of type integer, real, double precision, or complex, *e* must be an arithmetic constant expression.
- If *p* is of type character or logical, *e* must be a character constant expression or a logical constant expression, respectively.
- If a named constant is used in the constant expression *e*, it must be previously defined in the same **PARAMETER** or a preceding **PARAMETER** statement in the same program unit.
- A symbolic name of a constant must be defined only once in a **PARAMETER** statement within a program unit.
- The data type of a named constant must be specified by a type statement or **IMPLICIT** statement before its first appearance in a **PARAMETER** statement if a default implied type is not to be assumed for that symbolic name.
- Character symbolic named constants must be specified as type character in a **CHARACTER** statement, or the first letter of the name must appear in an **IMPLICIT** statement with the type **CHARACTER**. Specification must be made before the definition of the name in the **PARAMETER** statement.
- Once a symbolic name is defined, it can be used as a primary in any subsequent expressions or **DATA** statements in that program unit.
- The functions **IAND**, **IOR**, **NOT**, **IEOR**, **ISHFT**, **LGE**, **LGT**, **LLE**, and **LLT** with constant operands can be specified in a logical expression.
- The function **CHAR** with a constant operand can be specified in a character expression.
- The functions **MIN**, **MAX**, **ABS**, **MOD**, **ICHAR**, **NINT**, **DIM**, **DPROD**, **CMPLX**, **CONJG**, and **IMAG** with constant operands can be specified in arithmetic expressions.
- Symbolic names cannot specify the character count for Hollerith constants.
- Symbolic constants can appear in a **FORMAT** statement only within the context of a general expression bounded by angle brackets (<>).
- Symbolic constants cannot appear as part of another constant except when forming the real or imaginary part of a complex constant.

Restrictions

A constant and a symbolic name for a constant are generally not interchangeable. For example, a symbolic name of an integer constant cannot be used as a length specification in a **CHARACTER** type statement without enclosing parentheses. For instance, **CHARACTER*(I)** is valid, but **CHARACTER*I** is not.

However, a symbolic name of a constant can be used to form part of another constant, such as a complex constant, by using an intrinsic function as shown below:

```
complex c
real r
parameter (r = 2.0)
parameter (c = cmplx(1.0,r))
```

Examples

The following statements declare that 1 is converted to 1E0, making **X** the name of a **REAL** constant:

```
REAL X
PARAMETER (X = 1)
```

The following example converts 3.14 to 3, making **I** the name of an **INTEGER** constant:

```
INTEGER I
PARAMETER (I = 3.14)
```

The following example assigns the constant value of .087769 to **interest_rate**:

```
REAL*4 interest_rate
PARAMETER (interest_rate = .087769)
```

The same result could be achieved using Format 2 as follows:

```
PARAMETER interest_rate = .087769
```

POINTER

The **POINTER** statement establishes pairs of variables and pointers where each pointer contains the address of its paired variable.

Syntax

`POINTER (`*p1*`,`*v1*`) [,(`*p2*`,`*v2*`) ...]`

where

v1 and *v2* are pointer−based variables.

p1 and *p2* are the corresponding pointers. The pointer integers are automatically typed that way by the compiler. The pointer−based variables can be of any type, including structures. Even if there is a size specification in the type statement, no storage is allocated when such a pointer−based variable is defined.

Rules for Use

- Once you have defined a variable as based on a pointer, you must assign an address to that pointer. Reference the pointer−based variable with standard Fortran, and the compiler does the referencing by the pointer. (Whenever your program references a pointer–based variable, that variable's address is taken from the associated pointer.) Provide an address of a variable of the appropriate type and size.

- You must provide a memory area of the right size, and assign the address to a pointer, usually with the normal assignment statement or data statement, because no storage is allocated when a pointer–based variable is defined.

Restrictions

- A pointer–based variable cannot be used as a dummy argument or in **COMMON**, **EQUIVALENCE**, **DATA**, or **NAMELIST** statements.
- A pointer–based variable cannot itself be a pointer.
- The dimension expressions for pointer–based variables must be constant expressions in main programs. In subroutines and functions, the same rules apply for pointer–based variables as for dummy arguments. The expression can contain dummy arguments and variables in **COMMON** statements. Any variable in the expressions must be defined with an integer value at the time the subroutine or function is called.

Example

The following program uses a **POINTER** statement:

```
pointer (ptr,v), (ptr2, v2)
character a*12, v*12, z*1, v2*12
data a/'abcdefghijkl'/
ptr = loc (a)
ptr = ptr +4
ptr2 = malloc (12)
v2 = a
z = v (1:1)
print *, z
z = v2(5:5)
print *, z
call free (ptr2)
end
```

PROGRAM

The **PROGRAM** statement defines a symbolic name for the main program.

Syntax

`PROGRAM` *pgm*

where *pgm* is a symbolic name of the main program, which cannot be the name of an external procedure, block data subprogram, or common block or a local name in the same program unit.

Rules for Use

- The **PROGRAM** statement is optional. However, it must be the first statement in the main program when used.
- The symbolic name must be unique for that executable program. It must not be the name of any entity within the main program or any subprogram, entry, or common block.

RECORD

The **RECORD** statement creates a record in the format specified by a previously declared **STRUCTURE** statement. The effect of a **RECORD** statement is comparable to that of an ordinary type declaration.

Syntax

```
RECORD /structure-name/record-name[,record-name]
       [,record-name]...[/structure-name/
       record-name[,record-name][,record-name]...] ...
```

where

structure−name is the name of a previously declared structure (see the description of the **STRUCTURE** statement in "STRUCTURE / UNION").

record−name is a variable, an array, or an array declarator.

Method of Operation

The *record−name* can be used in **COMMON** and **DIMENSION** statements but not in **DATA**, **EQUIVALENCE**, **NAMELIST**, or **SAVE** statements. Records created by the **RECORD** statement are initially undefined unless the values are defined in the related structure declaration.

Examples

In the following statements, the record **latest** has the format specified by the structure **weather**; **past** is an array of 1,000 records, each record having the format of the structure **weather**.

```
STRUCTURE /weather/
    integer              month, day, year
    character*40   clouds
    real                 rainfall
end structure
record /weather/ latest, past (1000)
```

Individual items in the structure can be referenced using *record−name* and the name of the structure item. For example

```
past(n).rainfall = latest.rainfall
```

where **n** represents a number from 1 to 1,000 specifying the target array element. See the description of the **STRUCTURE** statement in this chapter for an example of how to declare a structure format.

SAVE

The **SAVE** statement retains the values of variables and arrays after execution of a **RETURN** or **END** statement in a subprogram. Therefore, those entities remain defined for subsequent invocations of the subprogram.

Syntax

```
SAVE [a[,a]...]
```

where *a* is one of the following:

- A variable or array name
- A common block name, preceded and followed by slashes

Method of Operation

The **SAVE** statement prevents named variables, arrays, and common blocks from becoming undefined after the execution of a **RETURN** or **END** statement in a subprogram. Normally, all variables and arrays become undefined on exit from a subprogram, except when they are

- specified by a **SAVE** statement
- defined in a **DATA** statement
- used in an **EQUIVALENCE** statement
- contained in a blank common
- contained in a named common that is declared in the subprogram and in a calling program unit in **SAVE** statements

All variables and arrays declared in the main program maintain their definition status throughout the execution of the program. If a local variable or array is not in a common block and is specified in a**SAVE** statement, it has the same value when the next reference is made to the subprogram.

All common blocks are treated as if they had been named in a **SAVE** statement. All data in any common block is retained on exit from a subprogram.

Note: Default **SAVE** status for common blocks is an enhancement to Fortran 77. In Fortran 77, a common block named without a corresponding **SAVE** statement causes the variables and arrays in the named common block to lose their definition status on exit from the subprogram.

Rules for Use

- A **SAVE** statement without a list is treated as though all allowable entities from that program unit were specified on the list.

- The main program can contain a **SAVE** statement, but it has no effect.
- A given symbolic name can appear in only one **SAVE** statement in a program unit.

Restrictions

Procedure names and dummy arguments cannot appear in a **SAVE** statement. The names of individual entries in a common block are not permitted in a **SAVE** statement.

Examples

The following statements are examples of **SAVE** statements:

```
SAVE L, V
SAVE /DBASE/
```

STRUCTURE / UNION

The **STRUCTURE** statement defines a record structure that can be referenced by one or more **RECORD** statement.

Syntax (General)

```
STRUCTURE [/structure–name/] [field–names]
    [field–definition]
    [field–definition] ...
END STRUCTURE
```

where

structure–name identifies the structure in a subsequent **RECORD** statement. Substructures can be established within a structure by means of either a nested **STRUCTURE** declaration or a **RECORD** statement.

field–names (for substructure declarations only) one or more names having the structure of the substructure being defined.

field–definition can be one or more of the following:

- Typed data declarations, which can optionally include one or more data initialization values.
- Substructure declarations (defined by either **RECORD** statements or subsequent **STRUCTURE** statements).
- **UNION** declarations, which are mapped fields defined by a block of statements. The syntax of a **UNION** declaration is described below.
- **PARAMETER** statements, which do not affect the form of the structure.

UNION Declaration Syntax

A **UNION** declaration is enclosed between **UNION** and **END UNION** statements, which contain two more map declarations. Each map declaration is enclosed between **MAP** and **END MAP** statements.

```
UNION
   MAP
           [field−definition]  [field−definition]  ...
   END MAP
   MAP
           [field−definition]  [field−definition]  ...
   END MAP
   [MAP
           [field−definition]  [field−definition]  ...
   END MAP] ...
 END UNION
```

Method of Operation

- Typed data declarations (variables or arrays) in structure declarations have the form of normal Fortran typed data declarations. Data items with different types can be freely intermixed within a structure declaration.
- Unnamed fields can be declared in a structure by specifying the pseudo name **%FILL** in place of an actual field name. You can use this mechanism to generate empty space in a record for purposes such as alignment.
- All mapped field declarations that are made within a **UNION** declaration share a common location within the containing structure. When initializing the fields within a **UNION**, the final initialization value assigned overlays any value previously assigned to a field definition that shares that field.

Examples (General)

```
structure /weather/
    integer             month, day, year
    character*20   clouds
    real                 rainfall
end structure
record /weather/ latest
```

In the preceding example, the **STRUCTURE** statement produces the storage mapping, shown in Figure 4−3, for the *latest* specification in the **RECORD** statement.

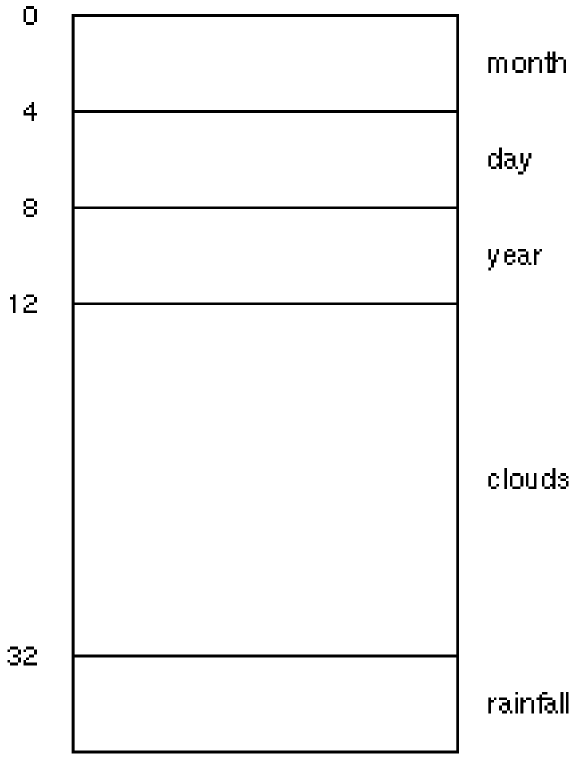

Figure 4–3 Logical Representation of a STRUCTURE Statement

The following gives an example of initializing the fields within a structure definition block:

```
   program weather
   structure /weather/
          integer*1     month /08/, day /10/, year /89/
          character*20  clouds /' overcast'/
          real   rainfall /3.12/
   end structure
   record /weather/ latest
   print *, latest.month, latest.day, latest.year,
 +  latest.clouds, latest.rainfall
```

The above example prints the following:

```
8 10 89 overcast  3.120000
```

Examples (UNION)

```
   program writedate
   structure /start/
          union
                  map
```

```
                        character*2 month
                        character*2 day
                        character*2 year
                 end map
                 map
                        character*6 date
                 end map
          end union
    end structure
    record /start/ sdate
    sdate.month =       '08'
    sdate.day =   '10'
    sdate.year =  '89'
    write (*, 10) sdate.date
10  format (a)
    stop
    end
```

In the above example, text is written to the standard I/O device as follows:

```
081089
```

VOLATILE

The **VOLATILE** statement prevents the compiler from optimizing specified variables, arrays, and common blocks of data.

Syntax

`VOLATILE` *volatile−items*

where *volatile−items* is one or more names of variables, common blocks, or arrays, each separated by a comma

For more information on optimization, refer to the *IRIX System Programming Guide* and the *f77(1)* manual page.

Chapter 5

Assignment and Data Statements

This chapter contains the following subsections:

- "Arithmetic Assignment Statements"
- "Logical Assignment Statements"
- "Character Assignment"
- "Aggregate Assignment"
- "ASSIGN"
- "Data Initialization"
- "Implied−DO Lists"

Assignment statements assign values to variables and array elements. Data statements and implied−**DO** lists in data statements are used to initialize variables and array elements.

The five types of Fortran assignment statements are

- arithmetic
- logical
- character
- aggregate
- statement label

This chapter explains how to use each of these statements. Each type is discussed in detail in the following sections.

Arithmetic Assignment Statements

An arithmetic assignment statement assigns the value of an arithmetic expression to a variable or array element of type **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, or **DOUBLE COMPLEX**. The form of an arithmetic statement is

v = *e*

where

v	is the name of an integer, real, double−precision, complex, or double−complex type variable or array element.
e	is an arithmetic expression.

When an arithmetic assignment statement is executed, the expression *e* is evaluated and the value obtained replaces the value of the entity to the left of the equal sign.

The values *v* and *e* need not be of the same type; the value of the expression is converted to the type of the variable or array element specified. Table 5−1lists the type conversion rules.

Table 5−1 Type Conversion Rules

Declaration	Function Equivalent
INTEGER	INT(*e*)
REAL	REAL(*e*)
DOUBLE PRECISION	DBLE(*e*)
COMPLEX	CMPLX(*e*)
DOUBLE COMPLEX	DCMPLX(*e*)

The following are examples of arithmetic assignment statements:

I = 4	Assign the value 4 to **I**
J = 7	Assign the value 7 to **J**
A = I*J+1	Assign the value 29 to **A**

Table 5−2gives the detailed conversion rules for arithmetic assignment statements. The functions in the second column of the table are intrinsic functions described in Chapter 10, "Statement Functions and Subprograms" and Appendix A, "Intrinsic Functions."

Table 5−2 Conversion Rules for Assignment Statements

Variable or Array Element (*v*)	INTEGER or LOGICAL Expression (*e*)	REAL Expression (*e*)	REAL *8 Expression (*e*)	REAL *16 Expression (*e*)	COMPLEX Expression (*e*)	COMPLEX *16 Expression (*e*)
INTEGER or LOGICAL	Assign *e* to *v*	Truncate *e* to integer and assign to *v*	Truncate *e* to integer and assign to *v*	Truncate *e* to integer and assign to *v*	Truncate real part of *e* to integer and assign to *v*	Truncate real part of *e* to integer and assign to *v*
REAL	Append fraction (.0) to *e* and assign to *v*	Assign *e* to *v*	Assign high−order portion of *e* to *v*; low−order portion of *e* is rounded	Assign high−order part of *e* to *v*; low−order part is rounded	Assign real part of *e* to *v*; imaginary part of *e* not used	Assign high−order part of real part of *e* to *v*; low−order portion of real part *e* is rounded
REAL *8	Append fraction (.0) to *e* and assign to *v*	Assign *e* to high−order portion of *v*; low−order portion of *v* is 0	Assign *e* to *v*	As above	Assign *e* to high−order portion of *v*; low−order portion of *v* is 0	Assign real part of *e* to *v*
REAL *16	As above	As above	As above	As above	As above	As above
COMPLEX	Append fraction to *e* and assign to real part of *v*; imaginary part of *v* is 0.0	Assign *e* to real part of *v*; imaginary part of *v* is 0.0	Assign high−order portion of *e* to real part of *v*; low−order portion of *e* is rounded; imaginary part of *v* is 0.0	Assign high−order portion of *e* to real part of *v*; low−order part is rounded; imaginary part of *v* is 0.0	Assign *e* to *v*	High−order parts of real and imaginary components of *e* are assigned to *v*; low−order parts are rounded
COMPLEX *16	Append fraction to *e* and assign to *v*; imaginary	Assign *e* to high−order portion of real	Assign *e* to real part of *v*; imaginary part is 0.0	As above	Assign *e* to high−order parts of *v*; low−order parts	Assign *e* to *v*

	to *v*; imaginary part of *v* is 0.0	portion of real part of *v*; imaginary part of *v* is 0.0	is 0.0			*v*; low−order parts of *v* are 0

Logical Assignment Statements

The logical assignment statement assigns the value of a logical expression to a logical variable or array element. It takes the form

v = *e*

where

v is the name of a logical variable or logical array element.

e is a logical expression.

When a logical assignment statement is executed, the value of the logical expression *e* is evaluated and replaces the value of the logical entity to the left of the equal sign. The value of the logical expression is either true or false.

Character Assignment

The character assignment statement assigns the value of a character expression to a character variable, array element, or substring. The form of a character assignment statement is

v = *e*

where

v is the name of a character variable, array element, or substring.

e is a character expression.

During the execution of a character string assignment statement, the character expression is evaluated and the resultant value replaces the value of the character entity to the left of the equal sign. None of the character positions being defined in *v* can be referenced in the evaluation of the expression *e*.

The entity *v* and character expression *e* can have different lengths. If the length of *v* is greater than the length of *e*, then the value of *e* is extended on the right with blank characters to the length of *v*. If the length of *e* is greater than the length of *v*, then the value of *e* is truncated on the right to the length of *v*.

The following is an example of character assignment:

```
CHARACTER U*5, V*5, W*7
U = 'HELLO'
V = 'THERE'
W(6:7) = V(4:5)
```

If an assignment is made to a character substring, only the specified character positions are defined. The definition status of character positions not specified by the substring remain unchanged.

Aggregate Assignment

An aggregate assignment statement assigns the value of each field of one aggregate to the corresponding field of another aggregate. The aggregates must be declared with the same structure. The form of an aggregate assignment statement is

v = *e*

where *v* and *e* are aggregate references declared with the same structure.

See the "Records" section and "Record and Field References" subsections in Chapter 2, "Constants and Data Structures," for more information.

ASSIGN

The **ASSIGN** statement assigns a statement label to an integer variable and is used in conjunction with an assigned **GOTO** statement or an I/O statement. The form of a statement label assignment statement is

`ASSIGN` *s* `TO` *e*

where

s	is a statement label of an executable statement or a **FORMAT** statement that appears in the same program unit as the **ASSIGN** statement.
e	is an integer variable name.

A statement label assignment by the **ASSIGN** statement is the only way of defining a variable with a statement label value. A variable defined with a statement label value may be used only in an assigned **GOTO** statement or as a format identifier in an I/O statement. The variable thus defined must not be referenced in any other way until it has been reassigned with an arithmetic value.

An integer variable that has been assigned a statement label value can be redefined with the same statement label, a different statement label, or an arithmetic integer variable.

Examples using the **ASSIGN** statement are shown below:

Example 1

```
ASSIGN 100 TO kbranch
    .
    .
    .
GO TO kbranch
```

Example 2

```
ASSIGN 999 TO ifmt
999 FORMAT(f10.5)
    .
    .
```

```
     .
READ (*, ifmt) x
     .
     .
     .
WRITE (*, fmt = ifmt) z
```

Data Initialization

Variables, arrays, array elements, and substrings can be initially defined using the **DATA** statement or an implied-**DO** list in a **DATA** statement. The **BLOCK DATA** subprogram is a means of initializing variables and arrays in named common blocks and is discussed in Chapter 4, "Specification Statements."

Entities not initially defined or associated with an initialized entity are undefined at the beginning of the execution of a program. Uninitialized entities must be defined before they can be referenced in the program.

Implied−DO Lists

The implied-**DO** list initializes or assigns initial values to elements of an array.

Syntax

(*dlist*, *i* = *e1*, *e2* [,*e3*])

where

dlist	is a list of array element names and implied-**DO** lists.
i	is a name of an integer variable, referred to as the *implied−DO variable* It is used as a control variable for the iteration count.
e1	is an integer constant expression specifying an initial value.
e2	is an integer constant expression specifying a limit value.
e3	is an integer constant expression specifying an increment value.

e1, *e2*, and *e3* are as defined in **DO** statements.

Method of Operation

An iteration count and the values of the implied-**DO** variable are established from *e1*, *e2*, and *e3* exactly as for a **DO**-loop, except that the iteration count must be positive.

When an implied-**DO** list appears in a **DATA** statement, the *dlist* items are specified once for each iteration of the implied-**DO** list with the appropriate substitution of values for any occurrence of the implied-**DO** variable. The appearance of an implied-**DO** variable in an implied-**DO** has no effect on the definition status of that variable name elsewhere in the program unit. For an example of an implied-**DO** list, see "DATA" in Chapter 4.

The *range* of an implied-**DO** list is *dlist*.

Rules

- The integer constant expressions used for *e1*, *e2*, and *e3* can contain implied-**DO** variables of other implied-**DO** lists.
- Any subscript expression in the list *dlist* must be an integer constant expression. The integer constant expression can contain implied-**DO** variables of implied-**DO** lists that have the subscript expression within their range.

Chapter 6

Control Statements

This chapter contains the following subsections:

Control statements affect the normal sequence of execution in a program unit. They are described in alphabetic order in this chapter and summarized below.

CALL	References a subroutine program in a calling program unit.
CONTINUE	Has no operational function; usually serves as the terminal statement of a**DO** loop.
DO	Specifies a controlled loop, called a **DO** loop, and establishes the control variable, indexing parameters, and range of the loop.
DO WHILE	Specifies a **DO** loop based on a test for true of a logical expression.
ELSE	Used in conjunction with the block **IF** or **ELSE IF** statements.
ELSE IF	Used optionally with the block **IF** statement.
END	Indicates the end of a program unit.

END DO Defines the end of an indexed **DO** loop or a **DO WHILE** loop.

END IF Has no operational function; serves as a point of reference like a **CONTINUE** statement in a **DO** loop.

GO TO (Unconditional)
Transfers program control to the statement identified by the statement label.

GO TO (Computed)
Transfers control to one of several statements specified, depending on the value of an integer expression.

GO TO (Symbolic name)
Used in conjunction with an **ASSIGN** statement to transfer control to the statement whose label was last assigned to a variable by an assign statement.

IF (Arithmetic) Allows conditional branching.

IF (Branch logical)
Allows conditional statement execution.

IF (Test Conditional)
Allows conditional execution of blocks of code. The block **IF** can contain **ELSE IF** statements for further conditional execution control. The block **IF** ends with the **END IF**.

PAUSE Suspends an executing program.

RETURN Returns control to the referencing program unit. It can appear only in a function or subroutine program.

STOP Terminates an executing program.

CALL

The **CALL** statement references a subroutine subprogram in a calling program unit.

Syntax

`CALL` *sub*`[( [`*a*`[,`*a*`]...] )]`

where

sub is the symbolic name of the subroutine.

a is an actual argument, an expression, array name, array elements, record elements, record arrays, record array elements, Hollerith constants, or an alternate return specifier of the form **s*, where *s* is a statement label, or **&***s*, where *s* is a statement label.

Method of Operation

A **CALL** statement evaluates the actual arguments, association of the actual arguments with the corresponding dummy arguments, and execution of the statements in the subroutine. Return of control from the referenced subroutine completes the execution of the **CALL** statement.

Rules for Use

- The actual arguments *a* form an argument list and must agree in order, number, and type with the corresponding dummy arguments in the referenced subroutine.
- A subroutine that has been defined without an argument can be referenced by a **CALL** statement of the following forms:

  `CALL` *sub*
  `CALL` *sub*`()`

- If a dummy procedure name is specified as a dummy argument in the referenced subroutine, then the actual argument must be an external procedure name, a dummy procedure name, or one of the allowed specific intrinsic names. An intrinsic name or an external procedure name used as an actual argument must appear in an **INTRINSIC** or **EXTERNAL** statement, respectively.
- If an asterisk is specified as a dummy argument, an alternate return specifier must be supplied in the corresponding position in the argument list of the **CALL** statement.
- If a Hollerith constant is used as an actual argument in a **CALL** statement, the corresponding dummy argument must not be a dummy array and must be of arithmetic or logical data type. This rule is an exception to the first rule above.
- A subroutine can call itself directly or indirectly (recursion).

Note: Recursion is an extension to Fortran 77. Fortran 77 does not allow a subroutine to reference itself.

Example

In the following example, the main routine calls **PAGEREAD**, passing the parameters **LWORDCOUNT** , **PAGE**, and **NSWITCH**. After execution of **PAGEREAD**, control returns to the main program, which stops.

```
   program MakeIndex
   character*50 page
   dimension page (100)
   nswitch = 0
111       lwordcount = inwords1*2
*
   call pageread (lwordcount,page,nswitch)
   stop
*
   subroutine pageread (lwordcount,page,nswitch)
   character*50 page
   dimension page (100)
```

```
   icount = 100
        .
        .
        .
   end
*
```

CONTINUE

The **CONTINUE** statement has no operational function. It usually serves as the terminal statement of a **DO** loop.

Syntax

```
CONTINUE
```

Method of Operation

When a **CONTINUE** statement that closes a **DO** loop is reached, control transfer depends on the control variable in the **DO** loop. In this case, control will either go back to the start of the **DO** loop, or flow through to the statement following the **CONTINUE** statement. (See Item 3, "Loop Control Processing," in "DO" for full information about control of **DO** loops.)

Example

In the following example, the **DO** loop is executed 100 times, and then the program branches to statement 50 (not shown).

```
    iwordcount = 100
    do 25, i= 1,lwordcount
    read (2, 20, end=45) word
 20 format (A50)
 25 Continue
 *
    goto 50
```

DO

The **DO** statement specifies a controlled loop, called a **DO** loop, and establishes the control variable, indexing parameters, and range of the loop.

Syntax

DO [*s*] [,]*i* = *e1*, *e2* [, *e3*]

where

s is a statement label of the last executable statement in the range of the **DO** loop. This statement is called the terminal statement of the **DO** loop. *s* can be omitted. If *s* is omitted, the loop must be terminated with an **END DO** statement. On completion of the loop, execution resumes with the first statement following the **END DO** statement.

i is a name of an integer, real, or double–precision variable, called the **DO** variable.

e1 is an integer, real, or double–precision expression that represents the initial value given to the **DO** variable.

e2 is an integer, real, or double–precision expression that represents the limit value for the **DO** variable.

e3 is an integer, real, or double–precision expression that represents the increment value for the **DO** variable.

Method of Operation

The range of a **DO** loop consists of all executable statements following the statement, up to and including the terminal statement of the **DO** loop. In a **DO** loop, the executable statements that appear in the **DO** loop range are executed a number of times as determined by the control parameters specified in the **DO** statement.

The execution of a **DO** loop involves the following steps:

1. **Activating the DO loop**. The **DO** loop is activated when the **DO** statement is executed. The initial parameter *m1*, the terminal parameter *m2*, and the incremental parameter *m3* are established by evaluating the expressions *e1*, *e2*, and *e3*, respectively. The expressions are converted to the type of the **DO** variable when the data types are not the same. The **DO** variable becomes defined with the value of the initial parameter *m1*. The increment *m3* cannot have a value of zero and defaults to the value 1 if *e3* is omitted.

2. **Computing the iteration count.** The iteration count is established from the following expression:

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

   The iteration count is zero in the following cases:

   ```
   m1 > m2     and  m3 > 0
   m1 << m2  and  m3 = 0
   ```

   If the initial value (*m1*) of the **DO** exceeds the limit value (*m2*), as in

   ```
   DO 10 I = 2,1
   ```

   the **DO** loop will not be executed unless the **-onetrip** compiler option is in effect. This option causes the body of a loop thus initialized to be executed once.

   Ordinarily, on encountering a **D** –loop whose upper or lower limit has already been reached, the compiler–generated code skips the loop. This action conforms with Fortran standards.

   To make Fortran 77 compatible with Fortran 66, the compiler allows you to generate code that performs a loop at least once, regardless of whether the upper or lower limit has already been reached. This is accomplished by specifying the **-onetrip** option as described in Chapter 1 of the *Fortran 77 Programmer's Guide*. This option is included for older programs written under the assumption that all loops would be performed at least once.

3. **Loop control processing.** This step determines if further execution of the range of the **DO** loop is required. Loop processing begins by testing the iteration count. If the iteration count is positive, the first statement in the range of the **DO** loop is executed. Normal execution proceeds until the terminal statement is processed. This constitutes one iteration of the loop. Incrementing is then required, unless execution of the terminal statement results in a transfer of control.

   If the iteration count is zero, the **DO** loop becomes inactive. Execution continues with the first executable statement following the terminal statement of the **DO** loop. If several **DO** loops share the same terminal statement, incremental processing is continued for the immediately containing **DO** loop.

4. **Incremental processing.** The value of the **DO** variable is incremented by the value of the incremental parameter *m3*. The iteration count is then decreased by one, and execution continues with loop control processing as described above.

   A **DO** loop is either active or inactive. A **DO** loop is initially activated when its **DO** statement is executed. Once active, a **DO** loop becomes inactive when one of the following occurs:

   - The iteration count is zero.
   - A **RETURN** statement within the **DO** loop range is executed.
   - Control is transferred to a statement outside the range of the **DO** loop but in the same program unit as the **DO** loop.
   - A **STOP** statement is executed or the program is abnormally terminated.

   Reference to a subprogram from within the range of the **DO** loop does not make the **DO** loop inactive except when control is returned to a statement outside the range of the **DO** loop.

   When a **DO** loop becomes inactive, the **DO** variable of the **DO** loop retains its last defined value.

Rules for Use

- You can nest **DO** loops but do not overlap them.
- If a **DO** statement appears within an **IF** block, **ELSE IF** block, or **ELSE** block, the range of the **DO** loop must be contained within that block.
- If a block **IF** statement appears within the range of a **DO** loop, the corresponding **END IF** statement must appear within the range of the **DO** loop.
- The same statement can serve as the terminal statement in two or more nested **DO** loops.

Restrictions

- Do not use the following statements for the statement labeled s in the **DO** loop:

Unconditional **GO TO**	**END IF**
Assigned **GO TO**	**RETURN**
Arithmetic **IF**	**STOP**
Block **IF**	**END**
ELSE IF	Another **DO** statement
ELSE	

- If the statement labeled s is a logical **IF** statement, it can contain any executable statement in its statement body, except the following:

DO statement	**END IF**
Block**IF**	**END**
ELSE IF	Another logical **IF** statement
ELSE	

- Except by the incremental process covered above, the **DO** variable must not be redefined during the execution of the range of the **DO** loop.
- A program must not transfer control into the range of a **DO** loop from outside the **DO** loop. When this happens, the result is indeterminate.
- When the **DO** variable is a floating–point variable, especially if the loop increment value*e3* cannot be represented exactly in floating–point form, the number of times the loop executes could be off by one due to floating–point arithmetic error.

Example

```
   DO 10, i = 1, 10
          D
          D
          D
 10       CONTINUE
          E
```

In the above example, the statements noted with a **D** following the **DO** statement are executed sequentially ten times, then execution resumes at the statement **E** following **CONTINUE**.

DO WHILE

The **DO WHILE** statement specifies a controlled loop, called a **DO** loop, based on a test for true of a logical expression.

Syntax

`DO [s[,]] WHILE (e)`

where

s is a statement label of the last executable statement in the range of the **DO** loop. This statement is called the terminal statement of the **DO** loop.

e is a logical expression.

If s is omitted, the loop must be terminated with an **END DO** statement.

Method of Operation

The **DO WHILE** statement tests the specified expression before each iteration (including the first iteration) of the statements within the loop. When the logical expression e is found to be true, execution resumes at the statement specified by the label *s.* If *e* is omitted, execution resumes at the first statement following the **END DO** statement.

ELSE

Use the **ELSE** statement in conjunction with the block **IF** or **ELSE IF** statements.

Syntax

```
ELSE
```

Method of Operation

An **ELSE** block is the code that is executed when an **ELSE** statement is reached. An **ELSE** block begins after the **ELSE** statement and ends before the **END IF** statement at the same **IF** level as the **ELSE** statement. (For details about the term **IF** level, refer to "IF (Test Conditional)".) As well as containing simple, executable statements, an **ELSE** block can be empty (contain no statements) or can contain embedded block IF statements. Do not confuse the **ELSE** block with the **ELSE** statement.

An **ELSE** statement is executed when the logical expressions in the corresponding block **IF** and **ELSE IF** statements evaluate to false. An **ELSE** statement does not evaluate a logical expression; the **ELSE** block is always executed if the **ELSE** statement is reached. After the last statement in the **ELSE** block is executed (and provided it does not transfer control), control flows to the **END IF** statement that closes that whole **IF** level.

Rules for Use

- Do not specify **ELSE IF** or **ELSE** statements inside an **ELSE** block at the same **IF** level.
- The **IF** level of the **ELSE** statement must be greater than zero; that is, there must be a preceding corresponding block **IF** statement.

Restrictions

- Enter an **ELSE** block only by executing the **ELSE** statement. Do not transfer control into the **ELSE** block from the outside.
- If an **ELSE** statement has a statement label, the label cannot be referenced by any statement.

Example

The following example shows an **ELSE** block.

```
IF (R) THEN
    A = 0
ELSE IF (Q) THEN
    A = 1
ELSE
    A = -1
END IF
```

ELSE IF

The **ELSE IF** statement is used optionally with the **IF** block statement.

Syntax

ELSE IF (*e*) THEN

where *e* is a logical expression.

Method of Operation

Two terms need to be defined to explain the **ELSE IF** statement: **ELSE IF** block (defined below) and **IF** level (defined in "IF (Branch Logical)").

An **ELSE IF** block is the code that is executed when the logical expression of an **ELSE IF** statement is true. An **ELSE IF** block begins after the **ELSE IF** statement and ends before the next **ELSE IF**, **ELSE**, or **END IF** statement at the same **IF** level as the **ELSE IF** statement. As well as containing simple, executable statements, an **ELSE IF** block can be empty (contain no statements) or can contain embedded block **IF** statements. Do not confuse the **ELSE IF** block with the **ELSE IF** statement.

When an **ELSE IF** statement is reached, the logical expression *e* is evaluated. If *e* is true, execution continues with the first statement in the **ELSE IF** block. If the **ELSE IF** block is empty, control is passed to the next **END IF** statement that has the same **IF** level as the **ELSE IF** statement. If *e* is false, program control is transferred to the next **ELSE IF**, **ELSE**, or **END IF** statement that has the same **IF** level as the **ELSE IF** statement.

After the last statement of the **ELSE IF** block is executed (and provided it does not transfer control), control is automatically transferred to the next **END IF** statement at the same **IF** level as the **ELSE IF** statement.

Rule for Use

The **IF** level of the **ELSE IF** statement must be greater than zero (there must be a preceding corresponding block **IF** statement).

Restrictions

- Do not transfer control into an **ELSE I** block from outside the **ELSE IF** block.
- No statement can reference the statement label (if any) of an **ELSE IF** statement. The only way to reach an **ELSE IF** statement is through its **IF** block statement.

Example

The following example shows an **ELSE IF** block.

```
IF(R) THEN
    A = 0
ELSE IF (Q) THEN
    A = 1
END IF
```

END

The **END** statement designates the end of a program unit.

Syntax

```
END
```

Method of Operation

An **END** statement in a main program has the same effect as a **STOP** statement: it terminates an executing program.

An **END** statement in a function or subroutine subprogram has the effect of a **RETURN** statement: it returns control to the referencing program unit.

Rules for Use

- An **END** statement cannot be the last statement in every program unit.
- Do not continue an **END** statement.

END DO

The **END DO** statement defines the end of a indexed **DO** loop or a **DO WHILE** loop.

Syntax

```
END DO
```

END IF

The **END IF** statement has no operational function. It serves as a point of reference like a **CONTINUE** statement in a **DO** loop.

Syntax

```
END IF
```

Rules for Use

- Every block **IF** statement requires an **END IF** statement to close that **IF** level. (**IF** level is described in "IF (Test Conditional)").
- The **IF** level of an **END IF** statement must be greater than zero (there must be a preceding corresponding **IF** block statement).

Example

See the example given with the description of the **ELSE** statement in "ELSE".

GO TO (Unconditional)

The unconditional **GO TO** statement transfers program control to the statement identified by the statement label.

Syntax

GO TO *s*

where *s* is a statement label of an executable statement appearing in the same program unit as the unconditional **GO TO**.

Example

The following statement transfers program control to statement 358 and normal sequential execution continues from there.

```
GO TO 358
```

GO TO (Computed)

The computed **GO TO** statement transfers control to one of several statements specified, depending on the value of an integer expression.

Syntax

GO TO (*s*[,*s*]...)[,]*i*

where

s is a statement number of an executable statement appearing in the same program unit as the computed **GO TO**.

i is an integer.

A noninteger expression can also be used for *i*. Non−integer expressions are converted to integers (fractional portions are discarded) before being used to index the list of statement labels.

Method of Operation

A computed **GO TO** statement evaluates the integer expression and then transfers program control to the specified statement

In the computed **GO TO** statement with the following form

```
GO TO (s1, s2, ... ,sn),i
```

if $i<1$ or $i>n$, the program control continues with the next statement following the computed **GO TO** statement; otherwise, program control is passed to the statement labeled *si*. Thus, if the value of the integer expression is 1, control of the program is transferred to the statement numbered *s1* in the list; if the value of the expression is 2, control is passed to the statement numbered *s2* in the list, and so on.

Rule for Use

The same statement label can appear more than once in the same computed **GO TO** statement.

Example

In the following example, the fifth list item is chosen because **KVAL** + 1 = 5. Program control is transferred to the statement labeled 350.

```
KVAL = 4
GO TO(100,200,300,300,350,9000)KVAL + 1
```

GO TO (Symbolic Name)

Use the symbolic **GO TO** statement in conjunction with an **ASSIGN** statement to transfer control to the statement whose label was last assigned to a variable by an **ASSIGN** statement.

Syntax

```
GO TO i [[,] (s [,s]...)]
```

where *i* is an integer variable name and *s* is a statement label of an executable statement appearing in the same program unit as the assigned **GO TO** statement.

Method of Operation

The variable *i* is defined with a statement label using the **ASSIGN** statement in the same program unit as the assigned **GO TO** statement. When an assigned **GO TO** is executed, control is passed to the statement identified by that statement label. Normal execution then proceeds from that point.

Rules for Use

- The same statement label can appear more than once in the same assigned **GO TO** statement.
- If a list in parentheses is present, the statement label assigned to *i* must be one of those in the list.

Example

```
GO TO KJUMP,(100,500,72530)
```

The value of **KJUMP** must be one of the statement label values: 100, 500, or 72530.

IF (Arithmetic)

The arithmetic **IF** statement allows conditional branching.

Syntax

IF (*e*) *s1*, *s2*, *s3*

where

e	is an arithmetic expression of type integer, real, or double–precision but not complex.
s1, *s2*, *s3*	are numbers of executable statements in the same program unit as the arithmetic **IF** statement.

Method of Operation

In the execution of an arithmetic **IF** statement, the value of the arithmetic expression *e* is evaluated. Control is then transferred to the statement numbered *s1*, *s2*, or *s3* if the value of the expression is less than zero, equal to zero, or greater than zero, respectively. Normal program execution proceeds from that point.

Rules for Use

You can use the same statement number more than once in the same arithmetic **IF** statement.

Example

Consider the following statement:

```
IF (A + B*(.5))500,1000,1500
```

If the expression **A + B*(.5)** is

- less than zero, control jumps to statement 500
- equal to zero, control jumps to statement 1000
- greater than zero, control jumps to statement 1500

IF (Branch Logical)

The branch logical **IF** statement allows conditional statement execution.

Syntax

IF (*e*) *st*

where

e	is a logical expression.
st	is any executable statement except **DO**, block **IF**, **ELSE IF**, **ELSE**, **END IF**, **END**, or another logical **IF** statement.

Method of Operation

A logical **IF** statement causes a Boolean evaluation of the logical expression. If the value of the logical expression is true, statement *st* is executed. If the value of the expression is false, execution continues with the next sequential statement following the logical **IF** statement.

Note that a function reference in the expression is allowed but might affect entities in the statement *st*.

Example

The following examples show branch logical **IF** statements.

```
IF(A .LE. B) A = 0.0
IF (M .LT. TOC) GOTO 1000
IF (J) CALL OUTSIDE(B,Z,F)
```

IF (Test Conditional)

The test conditional **IF** statement allows the conditional execution of blocks of code. The block **IF** can contain **ELSE** and **ELSE IF** statements for further conditional execution control. The block **IF** ends with the **END IF** statement.

Syntax

IF (*e*) THEN

where *e* is a logical expression.

Method of Operation

An **IF** block is the code that is executed when the logical expression of a block **IF** statement evaluates to true. An **IF** block begins after the block **IF** statement and ends before the **ELSE IF**, **ELSE**, or **END IF** statement that corresponds to the block **IF** statement. As well as containing simple, executable statements, an **IF** block can be empty (contain no statements) or can contain embedded block **IF** statements. Do not confuse the term **IF**–block with block **IF**.

Block **IF** statements and **ELSE IF** statements can be embedded, which can make figuring out which statements are in which conditional blocks confusing. The **IF** level of a statement determines which statements belong to which **IF−THEN−ELSE** block. Fortunately, the **IF** level of a statement can be found systematically. The **IF** level of a statement is

(*n1* − *n2*)

where (starting the count at the beginning of the program unit): *n1* is the number of block **IF** statements up to and including *s*, and *n2* is the number of **END IF** statements up to but *not* including *s*.

The **IF** level of every block **IF**, **ELSE IF**, **ELSE**, and **END IF** statement must be positive because those statements must be part of a block **IF** statement. The **IF** level of the **END** statement of the program unit must be zero because all block **IF** statements must be properly closed. The **IF** level of all other statements must either be zero (if they are outside all **IF** blocks) or positive (if they are inside an **I** −block).

When a block **IF** statement is reached, the logical expression *e* is evaluated. If *e* is true, execution continues with the first statement in the **IF** block. If the **IF** block is empty, control is passed to the next **END IF** statement that has the same **IF** level as the block **IF** statement. If *e* is false, program control is transferred to the next **ELSE IF**, **ELSE**, or **END IF** statement that has the same **IF** level as the block **IF** statement.

After the last statement of the **IF** block is executed (and provided it does not transfer control), control is automatically transferred to the next **END I***F* statement at the same **IF** level as the block **IF** statement.

Restriction

Control cannot be transferred into an **IF** block from outside the **IF** block.

Example

The following example shows a test conditional **IF** block.

```
IF(Q .LE. R) THEN
    PRINT ('Q IS LESS THAN OR EQUAL TO R')
ELSE
    PRINT ( 'Q IS GREATER THAN R')
END IF
```

PAUSE

The **PAUSE** statement suspends an executing program.

Syntax

PAUSE [*n*]

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

Method of Operation

A **PAUSE** statement without an *n* specification suspends execution of a program and issues the following message:

```
PAUSE statement executed
```

```
To resume execution, type go. Any other inputwill terminate job.
```

A **PAUSE** statement with an *n* specification displays the specified character constant or digits and issues the pause message. For example, the following statement

```
PAUSE "Console Check"
```

results in the following message being displayed:

```
PAUSE Console Check statement executed
```

```
To resume execution, type go. Any other inputwill terminate job.
```

If execution is resumed, the execution proceeds as though a **CONTINUE** statement were in effect.

At the time of program suspension, the optional digit string or character constant becomes accessible to the system as program suspension status information.

RETURN

The **RETURN** statement returns control to the referencing program unit. It can appear only in a function or subroutine subprogram.

Syntax

In a function subprogram

```
RETURN
```

In a subroutine subprogram

RETURN [*e*]

where *e* is an integer expression specifying an alternate return.

A noninteger expression can be used for *e*. Noninteger expressions are converted to integer, and the fractional portions discarded, before control is returned to the alternate return argument.

Method of Operation

A **RETURN** statement terminates the reference of a function or subroutine and transfers control back to the currently referenced program unit. In a function subprogram, the value of the function then becomes available to the referencing unit. In a subroutine, return of control to the referencing program unit completes execution of the **CALL** statement.

A **RETURN** statement terminates the association between the dummy arguments of the external procedure and the current actual arguments.

In a subroutine subprogram, if e is not specified in a **RETURN** statement or if the value of *e* is less than or greater than the number of asterisks in the **SUBROUTINE** or **ENTRY** statement specifying the currently referenced name, then control returns to the **CALL** statement that initiated the subprogram. Otherwise, the value of e identifies the *e*th asterisk in the dummy argument list of the currently referenced name. Control returns to the statement identified by the alternate return specifier in the **CALL** statement that is associated with the *e*th asterisk in the dummy argument list.

The execution of a **RETURN** statement causes all entities in an external procedure to become undefined except for entities that are

- specified in a **SAVE** statement
- blank
- specified in a named common.
- initialized in a **DATA** statement that has neither been redefined nor become undefined

STOP

The **STOP** statement terminates an executing program.

Syntax

```
STOP [n]
```

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

Method of Operation

The **STOP** statement terminates an executing program. If *n* is specified, the digit string or character constant becomes accessible to the system as program termination status information.

Chapter 7

Input/Output Processing

This chapter contains the following subsections:

- "Records"
- "I/O Statements"
- "Files"
- "Methods of File Access"
- "Units"

Input statements copy data from external media or from an internal file to internal storage. This process is called *reading*. Output statements copy data from internal storage to external media or to an internal file. This process is called *writing*.

The Fortran input/output (I/O) facilities give you control over the I/O system. This section deals primarily with the programmer−related aspects of I/O processing, rather than with the implementation of the processor−dependent I/O specifications.

See Chapter 1 of the *Fortran 77 Programmer's Guide* for information on extensions to Fortran 77 that affect I/O processing.

Records

A *record* is simply a sequence of values or characters. Fortran has three kinds of records:

- formatted
- unformatted
- endfile

A *record* is a logical concept; it does not have to correspond to a particular physical storage form. However, external media limitations can also limit the allowable length of records.

Formatted Records

A *formatted record* contains only ASCII characters and is terminated by a carriage−return or line−feed character. Formatted records are required only when the data must be read from the screen or a printer copy.

A formatted record can be read from or written to only by formatted I/O statements. Formatted records are measured in characters. The length is primarily a function of the number of characters that were written into the record when it was created, but it may be limited by the storage media or the CPU. A formatted record may have a length of zero.

Unformatted Records

Unformatted records contain sequences of values; both character and noncharacter are not terminated by any special character and cannot be accurately comprehended in their printed or displayed format. Generally, unformatted records use less space than formatted records and thus conserve storage space.

An unformatted record can be read from or written to only by unformatted I/O statements. Unformatted records are measured in bytes. That length is primarily a function of the output list used to write the record but may be limited by the external storage media or the CPU. An unformatted record can be empty.

Endfile Records

An *endfile record* marks the logical end of a data file. Thus, it can only be the last record of a file. An endfile record does not contain data and has no length. An endfile record is written by an **ENDFILE** statement.

When a program is compiled with **-vms_endfile**, an endfile record consists of a single character, Control D. In this case, several endfile records can exist in the same file and can be anywhere in the file. Reading an endfile record will result in an end−of−file condition being returned, but rereading the same file will read the next record, if any.

I/O Statements

The I/O statements that Fortran uses to transfer data can be categorized by how the data translated during the transfer, namely, as *formatted*, *list−directed*, and *unformatted I/O*.

Unformatted Statements

An unformatted I/O statement transfers data in the noncharacter format during an I/O operation. Unformatted I/O operations are usually faster than formatted operations, which translate data into character format.

In processing formatted statements, the system interprets some characters, for example, the line−feed character, as special controls and eliminates them from input records. Therefore, unformatted statements must be used when *all* characters in a record are required.

The absence of a format specifier denotes an unformatted data transfer statement, as shown by the **WRITE** statement in the following example:

```
    program MakeIndex
    character*12  word
    open (2, file='v',form='formatted')
    open (unit=10, status='new', file='newv.out",
   +       form='unformatted')
116 read (2,666, end=45) word
    write (10) word
    go to 116
45  close (10)
end
```

In the above example, formatted records are read into the variable *word* from the input file attached to unit 2 and then written unformatted to the output file attached to unit 10.

Formatted Statements

A *formatted I/O* statement translates all data to character format during a record transfer. The statement contains a *format specifier* that references a **FORMAT** statement; the **FORMAT** statement contains descriptors that determine data translation and perform other editing functions. Here is an example of two formatted **WRITE** statements:

```
    program makeindex
    character*18  message
    message = 'Hello world'
    write (6,100) message
    write (6,100) 'hello world'
100 format (a)
end
```

Note that both statements contain the format specifier 100, which references a format statement with an *A* character descriptor. (Chapter 9, "Format Specification," describes the descriptors in detail.) Both statements perform the same function, namely, writing the following message to the unit 6 device:

```
HELLO WORLD
```

List–Directed Statements

An I/O statement is *list directed* when an asterisk is used in place of a format specifier. A list–directed I/O statement performs the same function as a formatted statement. However, in translating data, a list–directed statement uses the declared data type rather than format descriptors in determining the format.

The following two list–directed **WRITE** statements perform the same function as the formatted **WRITE** statements in the example for formatted output above.

```
program makeindex
character*18  message
message = 'hello world'
write (6,*) message
write (6,*) 'hello world'
end
```

In this example, the variable *message* in the first **WRITE** statement determines that output is in character format; the character constant `Hello World` in the second statement makes this determination.

Files

A file is a sequence of records. The processor determines the set of files that exists for each executable program. The set of existing files can vary while the program executes. Files that are known to the operating system do not necessarily exist for an executable program at a given time. A file can exist and contain no records (all files are empty when they are created). I/O statements can be applied only to files that exist.

Files that have names are called *named files.* Names are simply character strings.

Every data file has a position. The position is used by I/O statements to tell which record to access and is changed when I/O statements are executed.

The terms used to describe the position of a file are

Initial point The point immediately before the first record.

Terminal point The point immediately after the last record.

Current record The record containing the point where the file is positioned. There is no current record if the file is positioned at the initial point (before all records) or at the terminal point (after all records) or between two records.

Preceding record

The record immediately before the current record. If the file is positioned between two records (so there is no current record), the preceding record is the record before the file position. The preceding record is undefined if the file is positioned in the first record or at the initial point.

Next record The record immediately after the current record. If the file is positioned between two records (so there is no current record), the next record is the record after the file position. The next record is undefined if the file position is positioned in the last record or at the terminal point.

This section discusses the two kinds of files: internal files and external files.

External Files

An *external file* is a set of records on an external storage medium (for example, a disk or a tape drive). A file can be empty, which means it can contain zero records.

Internal Files

An *internal file* is a means of transferring data within internal storage between character variables, character arrays, character array elements, or substrings.

An internal file is always positioned at the beginning of the first record before data transfer. Records are read from and written to by sequential access of formatted I/O statements only.

The following simple example shows how to use an internal file transfer to convert character and integer data.

```
program conversion
character*4   CharRep
integer       NumericalRep
```

```
NumericalRep = 10
C
C    example 1
C
     write (CharRep, 900) NumericalRep
900  format (i2)
     CharRep = '222'
C
C    example 2
C
     read (CharRep, 999) NumericalRep
999  format (i3)
     end
```

In the first example, the contents of **NumericalRep** are converted to character format and placed in **CharRep**. In the second example, the contents of **CharRep** are converted to integer format and placed in **NumericalRep**.

Methods of File Access

The following methods of file access are supported:

- sequential
- direct
- keyed

External files can be accessed using any of the above methods. The access method is determined when the file is opened or defined.

Fortran 77 requires that internal files must be accessed sequentially.

As an extension, the use of internal files in both formatted and unformatted I/O operations is permitted.

Sequential Access

A file connected for *sequential access* has the following properties:

- For files that allow only sequential access, the order of the records is simply the order in which they were written.
- For files that also allow direct access, the order of files depends on the record number. If a file is written sequentially, the first record written is record number 1 for direct access, the second written is record number 2, and so on.
- Formatted and unformatted records cannot be mixed within a file.
- The last record of the file can be an endfile record.
- The records of a pure sequential file must not be read or written by direct−access I/O statements.

Direct Access

A file connected for *direct access* has the following properties:

- A unique *record number* is associated with each record in a direct−access file. Record numbers are positive integers that are attached when the record is written. Records are ordered by their record numbers.
- Formatted and unformatted records cannot be mixed in a file.
- The file must not contain an endfile record if it is direct access only. If the file also allows sequential access, an endfile record is permitted but will be ignored while the file is connected for direct access.
- All records of the file have the same length. When the record length of a direct−formatted file is one byte, the system treats the files as ordinary system files, that is, as byte strings in which each byte is addressable. A **READ** or **WRITE** request on such files consumes/produces bytes until satisfied, rather than restricting itself to a single record. Note that to produce a record length of one byte, the program must be compiled with the **-old_rl** option.
- Only direct−access I/O statements can be used for reading and writing records. An exception is made when sequential I/O statements are used on a direct−unformatted file, in which case the next record is assumed. List−directed formatting is not permitted on direct−access files.
- The record number cannot be changed once it is specified. A record can be rewritten, but it cannot be deleted.
- Records can be read or written in any order.

Keyed Access

A file connected for *keyed access* has the following properties:

- Only files having an indexed organization can be processed using the keyed−access method.
- A unique character or integer value called a *key* is associated with one or more fields in each record of the indexed access file. The fields are defined when the file is created with an **OPEN** statement. Each **READ** statement contains a key to locate the desired record in the indexed file.
- You can intermix keyed access and sequential access on the same opened file.

Units

Files are accessed through *units.* A unit is simply the logical means for accessing a file. The file−unit relationship is strictly one to one: files cannot be connected to more than one unit and vice versa. Each program has a processor−dependent set of existing units. A unit has two states: connected and disconnected.

Connection of a Unit

A *connected unit* refers to a data file. A unit can be connected implicitly by the processor or explicitly by an **OPEN** statement. If a unit is connected to a file, the file is connected to the unit. However, a file can be connected and not exist. Consider, for example, a unit preconnected to a new file. A *preconnected unit* is already connected at the time the program execution begins. See the section on preconnected files in Chapter 1 of the *Fortran 77 Programmer's Guide* for these default connections.

Disconnection of a Unit

A unit can be *disconnected* from a file by a **CLOSE** statement specifying that particular unit.

Chapter 8

Input/Output Statements

This chapter contains the following subsections:

This chapter describes the statements that control the transfer of data within internal storage and between internal and external storage devices. It provides an overview of the Fortran I/O statements and gives syntax, rules, and examples for each.

This chapter also describes general rules that apply to data transfer statements.

Statement Summary

The I/O statements described in this chapter are grouped into the following classes:

- Data transfer statements, which transfer information between two areas of internal storage or between internal storage and an external file. The seven types are
  - **READ**
  - **DELETE**
  - **UNLOCK**
  - **ACCEPT**
  - **WRITE**
  - **REWRITE**
  - **PRINT** or **TYPE**
- Auxiliary statements, which explicitly open or close a file, provide current status information about a file or unit or write an endfile record. The four types are
  - **OPEN**
  - **CLOSE**
  - **INQUIRE**
  - **ENDFILE**
- File positioning statements, which position data files to the previous record or to the initial point of a file. These statements apply only to external files. They are
  - **BACKSPACE**
  - **REWIND**
- Statements that provide compatibility with earlier versions of Fortran. They are included to permit the older Fortran programs to be compiled and exist on the same system as standard Fortran 77 programs. The statements include the following:
  - **ENCODE**
  - **DECODE**
  - **DEFINE FILE**
  - **FIND**

The following sections describe the statements in the above summary in detail.

ACCEPT

The **ACCEPT** statement transfers data from the standard input unit to the items specified by the input list.

Syntax

ACCEPT *f* [,*iolist*]

where

f is the format specifier

iolist is an optional output list specifying where the data is to be stored.

See "Control Information List — cilist" and "Input/Output List — iolist" for a description of the *f* and *iolist* parameters.

Rules for Use

The **ACCEPT** statement specifies formatted input from the file associated with the system input unit; it cannot be connected to a user–specified input unit.

See "Data Transfer Rules" for additional rules.

Example

The following code transfers character data from the standard input unit into **x**.

```
      ACCEPT 3,x
 3    FORMAT (A)
```

BACKSPACE

The **BACKSPACE** statement positions a data file before the preceding record. It can be used with both formatted and unformatted data files.

Syntax

BACKSPACE *u*
BACKSPACE (*alist*)

where

u is an external unit identifier.

alist is a list of the following specifiers:

1.
   [UNIT =] *u* is a required unit specifier. *u* must be an integer expression that identifies the number of an external unit. If the keyword **UNIT =** is omitted, then *u* must be the first specifier in *alist*.

IOSTAT = *ios* — is an I/O status *specifier* that specifies the variable to be defined with a status value by the **BACKSPACE** statement. A zero value for *ios* denotes a no error condition, while a positive integer value denotes an error condition.

ERR = *s* — is an error specifier that identifies a statement number to which control is transferred when an error condition occurs during the execution of the **BACKSPACE** statement.

Note: An error message is issued if this statement references a file opened with an **ACCESS="KEYED"**, **ACCESS="APPEND"**, or a **FORM="SYSTEM"** specification.

Method of Operation

The unit specifier is required and must appear exactly once. The other specifiers are optional and can appear at most once each in the *alist*. Specifiers can appear in any order. For information about exceptions refer to "Unit Specifier— UNIT".

The **BACKSPACE** statement positions the file on the preceding record. If there is no preceding record, the position of the file is unchanged. If the preceding record is an endfile record, the file is positioned before the endfile record.

Examples

```
BACKSPACE M
BACKSPACE (6, IOSTAT=LP, ERR=998)
```

CLOSE

The **CLOSE** statement disconnects a particular file from a unit.

Syntax

CLOSE (*cilist*)

where *cilist* is a list of the following specifiers:

[UNIT =] *u* — is a required unit specifier. *u* must be an integer expression that identifies the number of an external unit. If the keyword **UNIT=** i*s* omitted, then *u* must be the first specifier in *cilist.*

IOSTAT=*ios* — is an I/O status specifier that specifies the variable to be defined with a status value by the **CLOSE** statement. A zero value for *ios* denotes a no error condition while a positive integer value denotes an error condition.

DISP[**OSE**]=*disposition*

Provides the same function as the like parameters in the **OPEN** statement. The *disposition* parameters in the file's **CLOSE** statement override the disposition parameters in its **OPEN** statement.

ERR=*s* — is an error specifier that identifies a statement number to which control is to be transferred when an error condition occurs during execution of the **CLOSE** statement.

STATUS='*sta*' is a file status specifier. *sta* is a character expression that, when any trailing blanks are removed, has a value of **KEEP** or **DELETE**. The status specifier determines the disposition of the file that is connected to the specified unit. **KEEP** specifies that the file is to be retained after the unit is closed. **DELETE** specifies that the file is to be deleted after the unit is closed. If a file has been opened for **SCRATCH** in an **OPEN** statement, then **KEEP** must not be specified in the **CLOSE** statement. If *iolist* contains no file status specifier, the default value is **KEEP**, except when the file has been opened for **SCRATCH**, in which case the default is **DELETE**.

Method of Operation

At the normal termination of an executable program, all units that are connected are closed. Each unit is closed with status **KEEP** unless the file has been opened for **SCRATCH** in an **OPEN** statement. In the latter case, the unit is closed as if with file status **DELETE**.

A **CLOSE** statement need not occur in the same program unit in which the file was opened. A **CLOSE** statement that specifies a unit that does not exist or has no file connected to it does not affect any file, and is permitted.

A unit that is disconnected by a **CLOSE** statement can be reconnected within the same executable program, either to the same file or to a different file. A file that is disconnected can be reconnected to the same unit or a different unit, provided that the file still exists.

Examples

```
CLOSE(UNIT=1,STATUS='KEEP')
CLOSE(UNIT=K,ERR=19,STATUS='DELETE')
```

DECODE

The **DECODE** statement transfers data between internal files, decoding the transferred data from character format to internal format.

Note: This statement provides primarily the same function as the **READ** statement using internal files, except that the input is read from a numeric scalar or array rather than a character string. This release does not support the concept of multiple records, and you must specify the record length. Where possible, use a **READ** statement instead of **DECODE** in new programs to make them compatible with different Fortran 77 operating environments.

Syntax

`DECODE (`*n*`,`*f*`,`*target*`[,ERR=`*s*`][,IOSTAT=`*rn*`]) [`*iolist*`]`

where

n is an integer expression specifying the number of characters to be translated to internal format.

f	is a format specifier (as described in "Format Specifier — FMT" in this chapter).
target	is a scalar reference or array indicating the destination of the characters after translation to external form.
ERR=*s*	See "Control Information List — cilist" in this chapter for an explanation of this parameter.
IOSTAT=*rn*	See "Control Information List — cilist" in this chapter for an explanation of this parameter.
iolist	is an optional list specifying the source data, as described in "Input/Output List — iolist" of this chapter.

Method of Operation

- The relationship between the I/O list and the format specifier is the same as for formatted I/O.
- The maximum number of characters transmitted is the maximum number possible for the *target* data type. If *target* is an array, the elements are processed in subscript order.

DEFINE FILE

The **DEFINE FILE** statement defines the size and structure of a relative file and connects it to a unit. It primarily provides the same function as the Fortran **OPEN** statement specifying **ACCESS='DIRECT'**.

Syntax

```
DEFINE FILE u (reccount,reclen,U,asvar)[,u (reccount,reclen,U,asvar)] ...
```

where

u	is an integer expression that identifies the number of an external unit that contains the file.
reccount	is an integer expression defining the number of records in the file.
reclen	is an integer expression specifying in word (two byte) units the length of each record.
U	specifies an unformatted (binary) file. *U* is always required and always in the same position, as shown in the above syntax.
asvar	is an associated integer variable indicating the next higher numbered record to be read or written. It is updated after each direct–access I/O operation.

Method of Operation

Only unformatted files can be opened with a **DEFINE FILE** statement. The file defined by *u* is assumed to contain fixed–length records of *recln* (two byte) words each. The records in the file are numbered 1 through *reccount*. The **DEFINE FILE** statement or equivalent **OPEN** statement must be executed before executing a **READ**, **WRITE**, or other direct–access statement. The first direct–access **READ** for the specified file opens an existing file; if the file does not exist, an error condition occurs. The first direct–access **WRITE** for the specified file opens the file and creates a new relative file.

DELETE

The **DELETE** statement removes a record from an indexed file. An error condition occurs if the file is not indexed.

Syntax

`DELETE [UNIT=]`*unum*

or

`DELETE ([UNIT=]`*unum*`[,IOSTAT=`*rn*`][,ERR=`*s*`]`

where

UNIT=*unum*	is a unit or internal file to be acted on.
IOSTAT=*rn*	is the name of variable in which I/O completion status is posted.
ERR=s	is a statement label to which control is transferred after an error.

See "Control Information List — cilist" and "Input/Output List — iolist" for details on these parameters.

Method of Operation

The **DELETE** statement deletes the current record, which is the last record accessed on unit *u*.

Example

The following statement deletes the last record read in from the file connected to logical unit 10.

```
DELETE (10)
```

ENCODE

The **ENCODE** statement transfers data between internal files, encoding the transferred data from internal format to character format.

Note: This statement primarily provides the same function as the **WRITE** statement, using internal files. Except that the input is read from a numeric scalar or array rather than a character string, the concept of multiple records is not supported. The record length is user specified. Where possible, use a **WRITE** statement instead of **ENCODE** in new programs to make them compatible with different Fortran 77 operating environments.

Syntax

`ENCODE (`*n*`,`*f*`,`*target*`[,ERR=`*s*`][,IOSTAT=`*rn*`]) [`*iolist*`]`

where

n	is an integer expression specifying the number of characters to be translated to character format.
f	is a format specifier (as described in the "Format Specifier — FMT").
ERR=*s*	See "Control Information List — cilist" for an explanation of this parameter.
IOSTAT=*rn*	See "Control Information List — cilist" for an explanation of this parameter.
target	is a scalar reference or array indicating the destination of the characters after translation to external form.
iolist	is an optional list specifying the source data, as described in "Input/Output List — iolist".

Method of Operation

The relationship between the I/O list and the format specifier is the same as for formatted I/O. *target* is padded with blanks if fewer than *n* characters are transferred. The maximum number of characters transmitted is the maximum number possible for the *target* data type. If *target* is an array, the elements are processed in subscript order.

ENDFILE

The **ENDFILE** statement writes an endfile record as the next record of the file. It can be used with both unformatted and formatted data files.

Syntax

```
ENDFILE u
ENDFILE (alist)
```

where

u	is an external unit identifier
alist	is a list of the following specifiers:
[UNIT =]*u*	is a required unit specifier. *u* must be an integer expression that identifies the number of an external unit. If the keyword **UNIT=** is omitted, then *u* must be the first specifier in *alist*.
IOSTAT=*ios*	is an I/O status specifier that specifies the variable to be defined with a status value by the **ENDFILE** statement. A zero value for *ios* denotes a no error condition, while a positive integer value denotes an error condition.
ERR=*s*	is an error specifier that identifies a statement number to which control is transferred when an error condition occurs during the execution of the **ENDFILE** statement.

Note: An error message is issued if this statement references a keyed−access file.

Method of Operation

The unit specifier is required and must appear exactly once. The other specifiers are optional and can appear at most once each in the *alist*. Specifiers can appear in any order (for exceptions refer to "Unit Specifier — UNIT").

An **ENDFILE** statement writes an endfile record. The specified file is then positioned after the endfile record. If a file is connected for direct access, only those records before the endfile record are considered to have been written and thus can be read in subsequent direct–access connections to the file.

An **ENDFILE** statement for a file that is connected but does not exist creates the file.

After an **ENDFILE** statement, a **BACKSPACE** or **REWIND** statement must be used to reposition the file before the execution of any data transfer I/O statement.

Note: If the program is compiled with the **-vms_endfile** option, the file can still be written to after the endfile record.

Examples

The following statements are examples of **ENDFILE** statements.

```
ENDFILE 2
ENDFILE (2,IOSTAT=IE, ERR=1000)
```

FIND

The **FIND** statement positions a file to a specified record number and sets the associate variable number (defined in an **OPEN** or **DEFINE FILE** statement) to reflect the new position. It is functionally equivalent to a direct–access **READ** statement except that no *iolist* is specified and no data transfer takes place. The statement opens the file if it is not already open.

Syntax

FIND ([UNIT=]*u*,REC=*rn*[,ERR=*s*][,IOSTAT=*rn*])

where

u — is an integer expression that identifies the number of an external unit that contains the file. The number must refer to a relative file.

ERR=*s*, **IOSTAT=***rn*, **REC=***rn* — See "Control Information List — cilist" for an explanation of these parameters.

INQUIRE

The **INQUIRE** statement inquires about the properties of a particular named file or the file connected to a particular unit. There are two forms: inquire by file and inquire by unit.

Syntax

```
INQUIRE (FILE=fname, [DEFAULTFILE=fname ...,]inqlist)
INQUIRE ([UNIT=]u, inqlist)
```

where

FILE=*fname* is a file specifier. *fname* is a character expression that specifies the name of the file being queried. The named file need not exist or be connected to a unit.

DEFAULTFILE=*fname*

This parameter corresponds to the **DEFAULTFILE** parameter in an **OPEN** statement and is used to inquire about a file assigned a default name when it was opened. See "OPEN" for details.

[UNIT=]*u* is a unit specifier. *u* must be an integer expression that identifies the number of an external unit. The specified unit need not exist or be connected to a file. If the keyword **UNIT=** is omitted, then *u* must be the first specifier in *inqlist*.

inqlist is composed of one or more of the following specifiers, separated by commas:

ACCESS=*acc* *acc* is a character variable or character array element to be assigned a value by the **INQUIRE** statement. The value assigned describes the type of file access as shown in Table 8−1

Table 8−1 File Access Types

Value Assigned	File Access
SEQUENTIAL	Sequential
DIRECT	Direct
KEYED	Keyed
UNKNOWN	No connection

BLANK=*blnk* *blnk* is a character variable or character array element to be assigned a value by the **INQUIRE** statement. The value assigned describes the blank specifier for the file as shown in Table 8−2

Table 8−2 Blank Control Specifiers

Value of ***blnk***	Specifier
NULL	Null blank control, connected for formatted I/O
ZERO	Zero blank control
UNKNOWN	Not connected or not connected for formatted I/O

CARRIAGECONTROL=*ccspec*

ccspec is assigned one of the following carriage control specifications made in the **OPEN** statement for the file: **FORTRAN**, **LIST**, **NONE**, or **UNKNOWN**.

DIRECT=*dir* *dir* is a character variable or character array element to be assigned a value by the **INQUIRE** statement. *dir* is assigned the value **YES** if **DIRECT** is a legal access method for the file; it is assigned the value **NO** if **DIRECT** is not a legal access method. If the processor is unable to determine the access type, *dir* is assigned the value **UNKNOWN**.

ERR=*s* is an error specifier that identifies a statement number to which control is transferred when an error condition occurs during the execution of the **INQUIRE** statement.

EXIST=*ex* *ex* is a logical variable or logical array element to be assigned a value by the **INQUIRE** statement. *ex* is assigned the value **.TRUE.** if the specified unit or file exists; otherwise, *ex* is assigned the value **.FALSE.** . A unit exists if it is a number in the range allowed by the processor.

FORM=*fm* *fm* is a character variable or character array element to be assigned a value by the **INQUIRE** statement. The value assigned is the form specifier for the file as shown in Table 8−3

Table 8−3 Form Specifiers

Value for *fm*	Specifier
FORMATTED	Formatted I/O
UNFORMATTED	Unformatted I/O
UNKNOWN	Unit is not connected

FORMATTED=*fmt*

fmt is a character variable or character array element to be assigned a value by the **INQUIRE** statement.*fmt* is assigned the value **YES** if **FORMATTED** is a legal form for the file;*fmt* is assigned the value **NO** if **FORMATTED** is not a legal form. If the processor is unable to determine the legal forms of data transfer,*fmt* is assigned the value **UNKNOWN**.

IOSTAT=*ios* is an I/O status specifier that specifies the variable to be defined with a status value by the **INQUIRE** statement. A zero value for *ios* denotes a no error condition, while a positive integer value denotes an error condition.

KEYED=*keystat*

keystat is a character scalar memory reference assigned a value as shown in Table 8−4

Table 8−4 Keyed−Access Status Specifiers

keystat	Meaning
YES	Indexed file, keyed access allowed
NO	Keyed access not allowed
UNKNOWN	Access type undetermined

NAMED=*nmd* *nmd* is a logical variable or logical array element to be assigned a value by the **INQUIRE** statement. *nmd* is assigned the value **.TRUE.** if the file has a name. Otherwise, *nmd* is assigned the value **.FALSE.**.

NAME=*fn* *fn* is a character variable or character array element to be assigned a value by the **INQUIRE** statement. *fn* is assigned the name of the file if the file has a name. Otherwise, *fn* is undefined. If the **NAME** specifier appears in an **INQUIRE** by file statement, its value is not necessarily the same as the name given in the file specifier.

NEXTREC=*nr* *nr* is an integer variable or integer array element to be assigned a value by the

INQUIRE statement. *nr* is assigned the value *n* + 1, where *n* is the record number of the last record read or written for direct access on the specified unit or file. If the file is connected but no records have been read or written, *nr* is assigned the value 1. If the file is not connected for direct access, *nr* is assigned the value 0.

NUMBER=*num*
num is an integer variable or integer array element that is assigned a value by the **INQUIRE** statement. *num* is assigned the external unit identifier of the unit currently connected to the file. *num* is undefined if there is no unit connected to the file. This specifier cannot be used with an **INQUIRE** by unit statement (**INQUIRE** (*iulist*)).

OPENED=*od*
od is a logical variable or logical array element to be assigned a value by the **INQUIRE** statement. *od* is assigned the value **.TRUE.** if the file specified is connected to a unit or if the specified unit is connected to a file. Otherwise, *od* is assigned the value **.FALSE.**.

ORGANIZATION=*org*
org is a character scalar memory reference assigned the value of the file organization established when the file was opened; it has one of the following values: **SEQUENTIAL**, **RELATIVE**, **INDEXED**, or **UNKNOWN** (always assigned to unopened files).

RECL=*rcl*
rcl is an integer variable or integer array element to be assigned a value by the **INQUIRE** statement. *rcl* is assigned the value of the record length in number of characters for formatted files and in words for unformatted files. If there is no connection or if the connection is not for direct access, *rcl* becomes undefined.

RECORDTYPE=*rectype*
rectype is a character scalar memory reference assigned the value of the record type file established when the file was opened; it has one of the following values:**FIXED**, **VARIABLE**, **STREAM_LF**, or **UNKNOWN**.

SEQUENTIAL=*seq*
seq is a character variable or character array element to be assigned a value by the **INQUIRE** statement. *seq* is assigned the value **YES** if **SEQUENTIAL** is a legal access method for the file. **seq** is assigned the value **NO** if **SEQUENTIAL** is not a legal access method. If the processor is unable to determine the legal access methods, *seq* is assigned the value **UNKNOWN**.

UNFORMATTED=*unf*
unf is a character variable or character array element to be assigned a value by the **INQUIRE** statement. *unf* is assigned the value of **YES** if **UNFORMATTED** is a legal format for the file; *unf* is assigned the value **NO** if **UNFORMATTED** is not a legal format for the file. If the processor is unable to determine the legal form,*unf* is assigned the value **UNKNOWN**.

Method of Operation

Specifiers can be given in *iflist* or *iulist* in any order ("Unit Specifier— UNIT" lists exceptions).

An **INQUIRE** statement assigns values to the specifier variables or array elements *nmd*, *fn*, *seq*, *dir*, *fmt*, and *unf* only if the value of the file specifier *fname* is accepted by the processor and if a file exists by that name. Otherwise, these specifier variables become undefined. Each specifier can appear at most once in the *iflist* or *iulist*, and the list must contain at least one specifier.

An **INQUIRE** statement assigns values to the specifier variables or array elements *num*, *nmd*, *fn*, *acc*, *seq*, *dir*, *fm*, *fmt*, *unf*, *rcl*, *nr*, and *blnk* only if the specified unit exists and if a file is connected to it. Otherwise, these specifier variables become undefined. However, the specifier variables *ex* and *od* are always defined unless an error condition occurs. All inquiry specifier variables except *ios* become undefined if an error condition occurs during execution of an **INQUIRE** statement.

Examples

The following examples show **INQUIRE** statements.

```
INQUIRE (FILE='MYFILE.DATA',NUMBER=IU,RECL=IR)
INQUIRE (UNIT=6, NAME=FNAME)
```

OPEN

The **OPEN** statement creates files and connects them to units. It can create a preconnected file, create and connect a file, connect an existing file, or reconnect an already connected file. See "File Positions" in Chapter 1 of the Fortran 77 Programmer's Guide for information on the relative record position in a file after an **OPEN** is executed.

Syntax

`OPEN (`*olist*`)`

where *olist* is a list of the following specifiers, separated by commas:

[**UNIT=**] *u* — is a required unit specifier. *u* must be an integer expression that identifies the number of an external unit. If the keyword **UNIT=** is omitted, then the *u* must be the first specifier in *olist.*

IOSTAT=*ios* — is an I/O status specifier that identifies the variable to be defined with a status value by the **OPEN** statement. A zero value for *ios* denotes a no error condition, while a positive integer value denotes an error condition.

ERR=s — is an error specifier that identifies a statement number to which program control is to be transferred when an error condition occurs during execution of the **OPEN** statement.

FILE=*fname* — is a file specifier. *fname* is a character expression specifying the name of the external file to be connected. The file name must be a name allowed by the processor. *fname* can also be a numeric variable to which Hollerith data is assigned. A null character terminates the filename. Three VMS predefined system logical names—**SYS$INPUT**, **SYS$OUTPUT**, and **SYS$ERROR**—are supported. These names allow an **OPEN**

statement to associate an arbitrary unit number to standard input, standard output, and standard error, respectively, instead of the standard predefined logical unit numbers 5, 6, and 0.

ACCESS=*acc* is an access specifier. *acc* is a character expression that, when trailing blanks are removed, has one of the following values: **SEQUENTIAL**, **DIRECT**, **KEYED**, or **APPEND**.**SEQUENTIAL** specifies that the file is to be accessed sequentially. **DIRECT** specifies that the file is to be accessed by record number. If **DIRECT** is specified, *iolist* must also contain a record length specifier. If *iolist* does not contain an access specifier, the value **SEQUENTIAL** is assumed. **KEYED** specifies that the file is accessed by a key−field value.**APPEND** specifies sequential access so that, after execution of an **OPEN** statement, the file is positioned after the last record.

ASSOCIATEVARIABLE=*asva*

specifies direct access only. After each I/O operation, *asvar* contains an integer variable giving the record number of the next sequential record number in the file. This parameter is ignored for all access modes other than direct access.

BLANK=*blnk* is a blank specifier. *blnk* is a character expression that, when all trailing blanks are removed, has the value **NULL** (the default) or **ZERO**. **NULL** ignores blank characters in numeric formatted input fields. **ZERO** specifies that all blanks other than leading blanks are to be treated as zeros. If *iolist* does not contain a blank specifier, the value **NULL** is assumed.

CARRIAGECONTROL=*type*

type is a character expression that determines carriage−control processing as shown in Table 8−5

Table 8−5 Carriage−Control Options

Value of *type*	Meaning
FORTRAN	Standard Fortran interpretation of the first character
LIST	Single spacing between lines
NONE	No implied carriage control

LIST is the default for formatted files, and **NONE** is the default for unformatted files. When the **-vms_cc** option (refer to Chapter 1 of the Fortran 77 Programmer's Guide) is specified, **FORTRAN** becomes the default for the standard output unit (unit 6).

DEFAULTFILE=*fname*

fname is either a character expression specifying a path name or an alternate prefix filename for the opened unit. When specified, the full filename of the opened unit is obtained by concatenating the string specified by *fname* with either the string in the **FILE** parameter (if specified) or with the unit number (when **FILE** is absent). *fname* can also be a numeric variable to which Hollerith data is assigned. A null character terminates the filename.

DISP[OSE]=*disposition*

disposition is a character expression that designates how the opened file is to be effect on the closed file.

Table 8–6 Disposition Options

Value for ***disposition***	**File status after CLOSE**
KEEP	Retained (default)
SAVE	Same as KEEP
PRINT	Printed and retained.
PRINT/DELETE	Printed and deleted.
SUBMIT	Executed and retained
SUBMIT/DELETE	Executed and deleted

FORM=*fm* is a form specifier. *fm* is a character expression that, when all trailing blanks are removed has either the value **FORMATTED** or **UNFORMATTED**. The file opened with **FORMATTED** is connected for formatted I/O, and a file opened with **UNFORMATTED** is connected for unformatted I/O. The extensions **SYSTEM** and **BINARY** can also be used to specify the form of the file. A file opened with the **SYSTEM** specifier is unformatted and has no record marks. Data is written/read as specified by the I/O list with no record boundary, which is equivalent to opening a file with the **BINARY** specifier on the IRIS 3000 series. A file opened with **BINARY** allows unformatted binary records to be read and written using formatted **READ** and **WRITE** statements. This form is only needed if the **A** edit descriptor is used to dump out numeric binary data to the file.If *iolist* contains no form specifier, the default value is **FORMATTED** for sequential access files and **UNFORMATTED** for direct access files.

KEY=(*key1start*:*key1end*[: *type*] [, *key2start*:*key2end*[:*type*]]...)

defines the location and data type of one or more keys in an indexed record. The following rules apply to **KEY** parameters:

- At least one key (the primary key) must be specified when creating an indexed file.
- *type* is either **INTEGER** or **CHARACTER** (the default), defining the data type of the key.
- **INTEGER** keys must be specified with a length of 4.
- The maximum length of a key is 512 bytes.
- *key1start* and *key1end* are integers defining the starting and ending byte positions of the primary field, which is always required. *key2start* and *key2end* and subsequent specifications define the starting and ending positions of alternate fields, which are optional. There is no limit to the number of keys that can be specified.
- The sequence of the key fields determines the value in a key–of–reference specifier, **KEYID**, described in "Control Information List — cilist". **KEYID=0** specifies the field starting the *key1start* (primary) key; **KEYID=1** specifies the

field starting at *key2start*, and so forth.

- The **KEY** field must be specified only when an indexed file is created. The key specifications in the first **OPEN** remain in effect permanently for subsequent file openings. If **KEY** is specified when opening an existing file, the specifications must match those specified when the file was created.

MAXREC=*n* where *n* is a numeric expression defining the maximum number of records allowed in a direct−access file. If this parameter is omitted, no maximum limit exists.

RECL=*rl* is a record length specifier. *rl* is a positive integer expression specifying the length in characters or processor−dependent units for formatted and unformatted files, respectively. This specifier is required for direct−access files and keyed−access files; otherwise, it must be omitted.

READONLY specifies that the unit is to be opened for reading only. Other programs may open the file and have read−only access to it concurrently. If you do not specify this keyword, you can both read and write to the specified file.

RECORDSIZE=*rl*

has same effect as **RECL**.

RECORDTYPE=*rt*

when creating a file *rt* defines the type of records that the file is to contain; *rt* can be one of the following character expressions: **FIXED**, **VARIABLE**, or **STREAM_LF**. If **RECORDTYPE** is omitted, the default record type depends on the file type, as determined by the **ACCESS** and/or **FORM** parameters. The default types are shown in Table 8−7.

Table 8−7 Default Record Types

File Type	Record Type (Default)
Relative or indexed	FIXED
Direct−access sequential	FIXED
Formatted sequential access	STREAM_LF
Unformatted sequential access	VARIABLE

The following rules apply:

- If **RECORDTYPE** is specified, *rt* must be the appropriate default value shown in Table 8−7.
- When writing records to a fixed−length file, the record is padded with spaces (for formatted files) or with zeros (for unformatted files) when the output statement does not specify a full record.

SHARED ensures that the file is as up to date as possible by flushing each record as it is written.

STATUS=*sta* is a file status specifier. *sta* is a character expression that, ignoring trailing blanks, has one of the following values: **OLD** requires the **FILE=***fname* specifier, and it must exist. **NEW** requires the **FILE=***fname* specifier. The file is created by **OPEN**, and the file status is automatically turned to **OLD**. A file with the same name must not already exist. **SCRATCH** creates an unnamed file that is connected to the unit from **UNIT=** and will be deleted when that unit is closed by **CLOSE**. **DEFAULTFILE** can be used to specify a temporary directory to be used for opening the temporary file. Named files should not be used with **SCRATCH**. **UNKNOWN** meaning is processor dependent. See the *Fortran 77 Programmer's Guide* for more information. If the **STATUS** specifier is omitted, **UNKNOWN** is the default.

TYPE=*sta* is the same as **STATUS**.

Rules for Use

- Specifiers can be given in *iolist* in any order (for an exception, see the **UNIT** specifier on page 170).
- The unit specifier is required; all other specifiers are optional. The record−length specifier is required for connecting to a direct−access file.
- The unit specified must exist.
- An **OPEN** statement for a unit that is connected to an existing file is allowed. If the file specifier is not included, the file to be connected to the unit is the same as the file to which the unit is connected.
- A file to be connected to a unit that is not the same as the file currently connected to the unit has the same effect as a **CLOSE** statement without a file status specifier. The old file is closed, and the new one is opened.
- If the file to be connected is the same as the file to which the unit is currently connected, then all specifiers must have the same value as the current connection except the value of the **BLANK** specifier.
- See "Data Transfer Rules" for additional rules.

Examples

The following examples show the use of **OPEN** statements:

```
OPEN (1, STATUS='NEW')
OPEN (UNIT=1,STATUS='SCRATCH',ACCESS='DIRECT',RECL=64)
OPEN (1, FILE='MYSTUFF', STATUS='NEW',ERR=14,
 + ACCESS='DIRECT',RECL=1024)
OPEN (K,FILE='MAILLIST',ACCESS='INDEXED',FORM='FORMATTED',
+RECL=256,KEY=(1:20,21:30,31:35,200:256))
```

PRINT or TYPE

The **PRINT** (or **TYPE**) statement transfers data from the output list items to the file associated with the system output unit.

Syntax

PRINT *f* [, *iolist*]

where *f* is the format specifier and *iolist* is an optional output list specifying the data to be transferred as described in "Control Information List — cilist" and "Input/Output List — iolist".

TYPE is a synonym for **PRINT**.

Rules for Use

Use the **PRINT** statement to transfer formatted output to the system output unit. See "Data Transfer Rules" for additional rules.

Examples

The following examples show the use of **PRINT** and **TYPE** statements.

```
PRINT 10, (FORM (L), L=1,K+1)
PRINT *, X,Y,Z
TYPE *, ' VOLUME IS ',V,' RADIUS IS ',R
```

READ (Direct Access)

The direct−access **READ** statement transfers data from an external file to the items specified by the input list. Transfer occurs using the direct−access method. (See Chapter 7, "Input/Output Processing," for details about the direct access method.)

Syntax: Formatted

READ ([UNIT=]*unum*, REC=*rn*, *f* [,IOSTAT=*ios*] [,ERR=*s*]) [*iolist*]

Syntax: Unformatted

READ ([UNIT=]*unum*, REC=*rn*, [,IOSTAT=*rn*] [,ERR=*s*]) [*iolist*]

where

UNIT=*unum*	is a unit or internal file to be acted on.
f	is a format specifier.
REC=*rn*	is a direct−access mode. *rn* is the number of the record to be accessed.
IOSTAT=*rn*	is the name of variable in which I/O completion status is posted.
ERR=*s*	is the statement label to which control is transferred after an error.
iolist	specifies memory location where data is to be read.

See the "Control Information List — cilist" and "Input/Output List — iolist" for details on these parameters.

See "Control Information List — cilist", "Input/Output List — iolist", and Chapter 7, "Input/Output Processing," for more information on formatted and unformatted I/O.

READ (Indexed)

The indexed **READ** statement transfers data from an external indexed file to the items specified by the input list. Transfer occurs using the keyed access method. (See Chapter 7, "Input/Output Processing.")

Syntax: Formatted

READ[UNIT=]*unum*,*f*,KEY=*val*[,KEYID=*kn*][,IOSTAT=*rn*][,ERR=*s*])[*iolist*]

Syntax: Unformatted

READ ([UNIT=]*unum*,*key*[,*keyid*][,IOSTAT=*rn*] [,ERR=*s*]) [*iolist*]

where

UNIT=*unum*	is a unit or internal file to be acted upon.
f	is a format specifier.
KEY=*val*	is the value of the key field in the record to be accessed.
KEYID=*kn*	is the key reference specifier.
IOSTAT=*rn*	is the name of variable to which I/O completion status is posted.
ERR=*s*	is the statement label to which control is transferred after an error.
iolist	specifies memory location where data is read.

See "Control Information List — cilist" and "Input/Output List — iolist" for details on these parameters.

See "Data Transfer Rules" and Chapter 7, "Input/Output Processing," for more information on indexed I/O and the differences between formatted and unformatted I/O.

READ (Internal)

The internal **READ** statement transfers data from an internal file to internal storage.

Syntax: Formatted

READ ([UNIT=]*unum*,*f*[,IOSTAT=*rn*][,ERR=*s*][,END=*eof*]) [*iolist*]

Syntax: List–Directed

READ ([UNIT=]*unum*,* [,IOSTAT=*rn*][,ERR=*s*][,END=*eof*]) [*iolist*]

where

UNIT=*unum*	is a unit or internal file to be acted upon.
f	is a format specifier
*	is a list–directed input specifier.
IOSTAT=*rn*	is the name of variable in which I/O completion status is to be posted.

ERR=*s* is the statement label to which control is transferred after an error.

END=*eof* is the statement label to which control is transferred upon end−of−file.

iolist specifies memory location where data is to be read.

See "Control Information List — cilist" and "Input/Output List — iolist" for details on these parameters.

See "Data Transfer Rules" and Chapter 7, "Input/Output Processing," for more information on formatted and list−directed I/O. Chapter 7 also contains an example of I/O using internal files.

Note: The **DECODE** statement can also be used to control internal input. See "DECODE" for more information.

READ (Sequential)

The sequential **READ** statement transfers data from an external record to the items specified by the input list. Transfers occur using the sequential−access method or keyed−access method. (See Chapter 7, "Input/Output Processing.")

The four forms of the sequential **READ** statement are

- formatted
- list−directed
- unformatted
- namelist−directed

The following parameters apply to all four forms of the sequential **READ** statement:

[**UNIT=**]*unum*
is a unit or internal file to be acted upon.

f is a format specifier.

***** is a list−directed input specifier.

NML=[*group−name*]
is a namelist specifier. If the keyword **NML** is omitted, *group−name* must immediately follow *unum***.**

IOSTAT=*rn* is the name of the variable in which I/O completion status is posted.

ERR=*s* is the statement label to which control is transferred after an error.

END=*eof* is the statement label to which control is transferred on end of file.

iolist specifies memory location where data is read.

See "Control Information List — cilist" for details on these parameters.

Formatted READ (Sequential)

Syntax

```
READ ([UNIT=]unum,f[,IOSTAT=rn][,ERR=s] [,END=eof]) [iolist]
READ f[,iolist]
```

Method of Operation

A formatted **READ** statement transfers data from an external record to internal storage. It translates the data from character to binary format using the *f* specifier to edit the data.

List–Directed READ (Sequential)

Syntax

```
READ ([UNIT=]unum,*[,IOSTAT=rn][,ERR=s][,END=eof])[iolist]
READ f*[iolist]
```

Method of Operation

A list–directed**READ** statement transfers data from an external record to internal storage. It translates the data from character to binary format using the data types of the items in *iolist* to edit the data.

Rules for Use

- The external record can have one of the following values:
  - A constant with a data type of integer, real, logical, complex, or character. The rules given in Chapter 2, "Constants and Data Structures," define the acceptable formats for constants in the external record.
  - A null value, represented by a leading comma, two consecutive constants without intervening blanks, or a trailing comma.
  - A repetitive format *n*constant*, where *n* is a nonzero, unsigned integer constant indicating the number of occurrences of *constant*. *n** represents repetition of a null value.
- Hollerith, octal, and hexadecimal constants are not allowed.
- A value separator must delimit each item in the external record; a value separator can be one of the following:
  - one or more spaces or tabs
  - a comma, optionally surrounded by spaces or tabs
- A space, tab, comma, or slash appearing within a character constant are processed as part of the constant, not as delimiters.
- A slash delimits the end of the record and causes processing of an input statement to halt; the slash can be optionally surrounded by spaces and/or tabs. Any remaining items in *iolist* are unchanged after the **READ**.

- When the external record specified contains character constants, a slash must be specified to terminate record processing. If the external record ends with a blank, the first character of the next record processed follows immediately after the last character of the previous record.
- Each **READ** statement reads as many records as is required by the specifications in *iolist*. Any items in a record appearing after a slash are ignored.

Unformatted READ (Sequential)

Syntax

READ ([UNIT=]*unum*[,IOSTAT=*rn*][,ERR=*s*][,END=*eof*]) [*iolist*]

Method of Operation

An unformatted **READ** statement transfers data from an external record to internal storage. The **READ** operation performs no translation on read−in data. The data is read in directly to the items in *iolist*. The type of each data item in the input record must match that declared for the corresponding item in *iolist*.

When a sequential−unformatted **READ** is performed on a direct−access file, the next record in the direct−access file is assumed.

Rules for Use

- There must be at least as many items in the unformatted record as there are in *iolist*. Additional items in the record are ignored, and a subsequent **READ** accesses the next record in the file.
- The type of each data item in the i3nput record must match the corresponding data item in *iolist*.

Namelist−Directed READ (Sequential)

Syntax

READ (*unum*,NML=*group−name*[,IOSTAT=*rn*][,ERR=*s*][,END=*eof*]) [*iolist*]
READ *name*

Method of Operation

A namelist−directed **READ** statement locates data in a file using the group name in a **NAMELIST** statement (see Chapter 4, "Specification Statements.") It uses the data types of the items in the corresponding **NAMELIST** statement and the forms of the data to edit the data.

Figure 8−1 illustrates rules for namelist input data and shows its format.

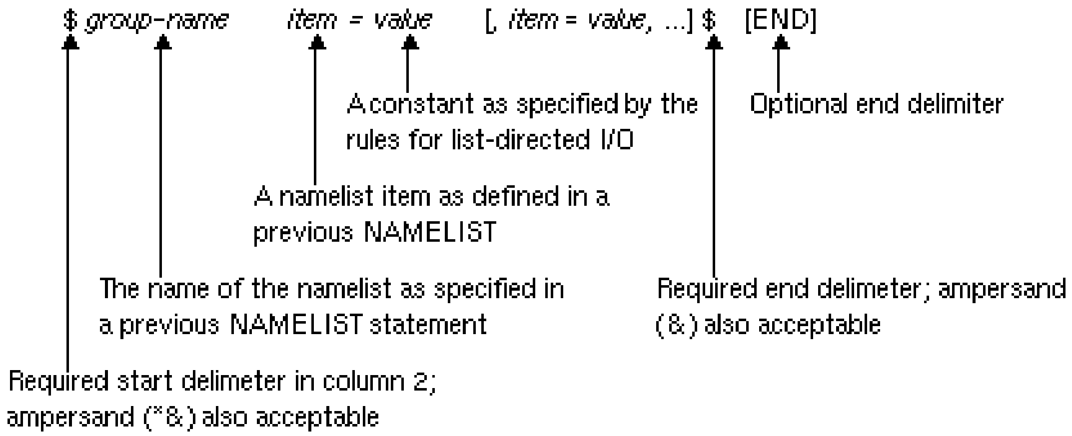

Figure 8–1 Namelist Input Data Rules

Rules for Use

- Both *group–name*and *item* must be contained within a single record.
- Spaces and/or tabs are not allowed within *group–name*or *item*. However, *item* can contain spaces or tabs within the parentheses of a subscript or substring specifier.
- The *value* item can be any of the values given under the first rule in the previous section, "List–Directed READ (Sequential)".
- A *value* separator must delimit each item in a list of constants. See the third and fourth rules in "List–Directed READ (Sequential)".
- A separator must delimit each list of value assignments. See the third rule in "List–Directed READ (Sequential)". Any number of spaces or tabs can precede the equal sign.
- When *value* contains character constants, a dollar sign ($) or ampersand (&) must be specified to terminate processing of the namelist input. If the namelist input ends with a blank, the first character of the next record processed follows immediately after the last character of the previous record.
- Entering a question mark (?) after a namelist–directed**READ** statement is executed causes the *group–name*and current values of the namelist items for that group to be displayed.
- You can assign input values in any order in the format *item*=*value*. Multiple–line assignment statements are allowed. Each new line must begin on or after column 2; column 1 is assumed to contain a carriage–control character. Any other character in column 1 is ignored.
- You can assign input values for the following data types: integer, real, logical, complex, and character. Refer to Table 5–2in Chapter 5 for the conversion rules when the data type of the namelist item and the assigned constant value do not match.
- Numeric–to–character and character–to–numeric conversions are not allowed.

- Constant values must be given for assigned values, array subscripts, and substring specifiers. Symbolic constants defined by a **PARAMETER** statement are not allowed.

Example

In the following example, the name of a file is read from the standard input into **filename**, the file is opened, and the first record is read. A branch is taken to statement 45 (not shown) when end of file is encountered.

```
   read (*,10) filename
10 format (a)
   open (2,file=filename)
   read (2, 20, end=45) word
20 format (A50)
```

See "Data Transfer Rules" and Chapter 7, "Input/Output Processing," for more information on formatted, list−directed unformatted, and namelist−directed I/O.

REWIND

The **REWIND** statement positions a file at its initial point. It can be used with both unformatted and formatted data files.

Syntax

REWIND *u*
REWIND (*alist*)

where

u	is an external unit identifier.
alist	is a list of the following specifiers:
[**UNIT =**] *u*	is a required unit specifier. *u* must be an integer expression that identifies the number of an external unit. If the keyword **UNIT=** is omitted, then *u* must be the first specifier in *alist*.
IOSTAT = *ios*	is an I/O status specifier that specifies the variable to be defined with a status value by the **REWIND** statement. A zero value for *ios* denotes a no error condition, while a positive integer value denotes an error condition.
ERR = *s*	is an error specifier that identifies a statement number to which control is transferred when an error condition occurs during the execution of the **REWIND** statement.

Method of Operation

The unit specifier is required and must appear exactly once. The other specifiers are optional and can appear at most once each in the *alist*. Specifiers can appear in any order (refer to"Unit Specifier— UNIT" for exceptions). The **REWIND** statement positions the specified file at its initial point. If the file is already at its initial point, the **REWIND** statement has no effect. It is legal to specify a **REWIND** statement for a file that is connected but does not exist, but the statement has no effect.

Examples

The following statements show examples of the **REWIND** statement.

```
REWIND 8
REWIND (UNIT=NFILE,ERR=555)
```

REWRITE

The **REWRITE** statement transfers data to an external indexed file from the items specified by the output list. The record transferred is the last record accessed from the same file using an indexed **READ** statement.

Syntax: Formatted

`REWRITE ([UNIT=]`*unum*`,`*f*`[,IOSTAT=`*rn*`][,ERR=`*s*`]) [`*iolist*`]`

Syntax: Unformatted

`REWRITE ([UNIT=]`*unum*`[,IOSTAT=`*rn*`][,ERR=`*s*`]) [`*iolist*`]`

where

[**UNIT=**]*unum*	is the unit or internal file to be acted on.
f	is a format specifier.
IOSTAT=*rn*	is the name of a variable in which I/O completion status is posted.
ERR=*s*	is a statement label to which control is transferred after an error.

See "Control Information List — cilist" and "Input/Output List — iolist" for details on these parameters.

Rules for Use

The **REWRITE** statement is supported for both formatted and unformatted indexed files. The statement provides a means for changing existing records in the file.

See "Data Transfer Rules" for additional rules.

Example

```
REWRITE (10), A,B,C
```

The previous statement rewrites the last record accessed to the indexed file connected to logical unit 10.

UNLOCK

The **UNLOCK** statement makes the last record read from an indexed file available for access by other users.

Syntax

```
UNLOCK [UNIT=]unum
UNLOCK ([UNIT=]unum[,IOSTAT=rn][,ERR=s]
```

where

UNIT=*unum* is a unit or internal file to be acted on.

IOSTAT=*rn* is the name of variable in which I/O completion status is posted.

ERR=*s* is the statement label to which control is transferred after an error.

See "Control Information List — cilist" for details on each of these parameters.

Method of Operation

After a record is read from an indexed file, it cannot be accessed by other users until an **UNLOCK** statement is executed, the record is rewritten, or a new record is read.

Example

The following statement unlocks the last record read in from the file connected to logical unit 10.

```
UNLOCK (10)
```

WRITE (Direct Access)

The direct–access **WRITE** statement transfers data from internal storage to an external indexed file using the direct–access method.

Syntax: Formatted

```
WRITE ([UNIT=]unum,REC=rn,f[,IOSTAT=rn][,ERR=s]) [iolist]
```

Syntax: Unformatted

```
WRITE ([UNIT=]unum,REC=rn[,IOSTAT=ios][,ERR=s]) [iolist]
```

where

[**UNIT=**]*unum* is a unit or internal file to be acted upon.

REC=*rn* is a direct–access mode. *rn* is the number of the record to be accessed.

f is a format specifier.

IOSTAT=*rn* is the name of variable in which I/O completion status is posted.

ERR=*s* is the statement label to which control is transferred after an error.

iolist specifies memory location from which data is written.

See "Control Information List — cilist" and "Input/Output List — iolist" for details on these parameters.

See "Data Transfer Rules" and Chapter 7, "Input/Output Processing," for more information on formatted and unformatted I/O.

Rules for Use

Execution of a **WRITE** statement for a file that does not exist creates the file.

WRITE (Indexed)

The indexed **WRITE** statement transfers data from internal storage to external records using the keyed–access method.

Syntax: Formatted

`WRITE ([UNIT=]`*unum*`,`*f*`[,IOSTAT=`*rn*`][,ERR=`*s*`]) [`*iolist*`]`

Syntax: Unformatted

`WRITE ([UNIT=]`*unum*`[,IOSTAT=`*rn*`][,ERR=`*s*`]) [`*iolist*`]`

where

[**UNIT=**]*unum*	is a unit or internal file to be acted on.
f	is a format specifier.
*	is the list–directed output specifier.
IOSTAT=*rn*	is the name of a variable in which I/O completion status is posted.
ERR=s	is a statement label to which control is transferred after an error.
iolist	specifies a memory location from which data is written.

See "Control Information List — cilist" and "Input/Output List — iolist" for details on these parameters.

See "Data Transfer Rules" and Chapter 7, "Input/Output Processing," for more information on formatted and unformatted I/O.

Rules for Use

Execution of a **WRITE** statement for a file that does not exist creates the file.

WRITE (Internal)

The internal **WRITE** statement transfers data to an external file or an internal file from the items specified by the output list.

Syntax: Formatted

`WRITE ([UNIT=]`*unum*`,`*f*`[,IOSTAT=`*ios*`][,ERR=`*s*`])[`*iolist*`]`

Syntax: List−directed

```
WRITE ([UNIT=]unum, *[,IOSTAT=rn][,ERR=s])[iolist]
```

where

[**UNIT=**]*unum*	is a unit or internal file to be acted on.
f	is a format specifier.
*	is the list−directed output specifier.
IOSTAT=*rn*	is the name of a variable in which I/O completion status is posted.
ERR=*s*	is the statement label to which control is transferred after an error.
iolist	specifies a memory location from which data is written.

See "Control Information List — cilist" and"Input/Output List — iolist" for details on these parameters.

See "Data Transfer Rules" and Chapter 7, "Input/Output Processing," for more information on formatted and list−directed I/O. Chapter 7 also contains an example of I/O using internal files.

Rules for Use

Execution of an internal **WRITE** statement for a file that does not exist creates the file.

Note: The **ENCODE** statement can also be used to control internal output. See the **ENCODE** statement description on page 161 for more information.

WRITE (Sequential)

The sequential **WRITE** statement transfers data to an external file or an internal file from the items specified by the output list.

The four types of sequential **WRITE** statements are

- formatted
- unformatted
- list−directed
- namelist−directed

Each of these statements is discussed in the following sections.

Execution of a **WRITE** statement for a file that does not exist creates the file.

Parameter Explanations

UNIT=*unum* is a unit or internal file to be acted on.

NML= *group−name*
is a namelist specifier.

f is a format specifier.

* is the list−directed output specifier.

REC=*rn* is a direct−access mode. *rn* is the number of the record to be accessed.

IOSTAT=*rn* is the name of a variable in which I/O completion status is posted.

ERR=*s* is a statement label to which control is transferred after an error.

iolist specifies a memory location from which data is written.

See "Control Information List — cilist" and "Input/Output List — iolist" for details on these parameters.

See "Data Transfer Rules" and Chapter 7, "Input/Output Processing," for more information on formatted, list−directed, and unformatted I/O.

Formatted WRITE (Sequential)

```
WRITE ([UNIT=]unum,f[,IOSTAT=rn][,ERR=s]) [iolist]
```

Method of Operation

A formatted **WRITE** statement transfers data from internal storage to an external record using sequential−access mode. The **WRITE** operation translates the data from binary to character format using the *f* specifier to edit the data.

Unformatted WRITE (Sequential)

```
WRITE ([UNIT=]unum[,IOSTAT=rn][,ERR=s]) [iolist]
```

Method of Operation

An unformatted **WRITE** statement performs no translation on read−in data. The data is read in directly to the items in *iolist*. The type of each data item in the input record must match that declared for the corresponding item in *iolist*.

When sequential−formatted **WRITE** is performed on a direct−access file, the next record in the file is assumed and the record is zero−padded to the end as if it were a direct, unformatted **WRITE**.

List−Directed WRITE

```
WRITE ([UNIT=]unum,*[,IOSTAT=rn][,ERR=s]) [iolist]
```

Method of Operation

A list−directed **WRITE** statement transfers data from internal storage to an external record using sequential−access mode. The **WRITE** operation translates the data from binary to character format using the data types of the items in *iolist* to edit the data.

Rules

- The item to be transferred to an external record can be a constant with a data type of integer, real, logical, complex, or character.
- The rules given in Chapter 2, "Constants and Data Structures,"define the acceptable formats for constants in the external record, except character constant. A character constant does not require delimiting apostrophes; an apostrophe within a character string is represented by one apostrophes instead of two.

Table 8−8shows the data types and the defaults of their output format.

Table 8−8 Default Formats of List−Directed Output

Data Type	Format Specification of Default Output
BYTE	L2
LOGICAL*1	I5
LOGICAL*2	L2
LOGICAL*4	L2
INTEGER*1	I5
INTEGER*2	I7
INTEGER*4	i12
REAL*4	1pg15.7e2
REAL*8	1pg24.16e2
COMPLEX	’(’,1pg15.7e2,’,’,1pg15.7e2,’)’
COMPLEX*16	’(’,1pg24.16e2,’,’,1pg24.16e2,’)’
CHARACTER**n*	A*n*, where *n* is the length of the character expression

- List−directed character output data cannot be read as list−directed input because of the use of apostrophes described above.
- A list−directed output statement can write one or more records. Position one of each record must contain a space (blank), which Fortran uses for a carriage−control character. Each value must be contained within a single record with the following exceptions:
  – A character constant longer than a record can be extended to a second record.
  – A complex constant can be split onto a second record after the comma.
- The output of a complex value contains no embedded spaces.
- Octal values, null values, slash separators, or the output of a constant or null value in the repetitive format *n*constant* or *n*z* cannot be generated by a list−directed output statement.

Namelist−Directed WRITE

Syntax

WRITE([UNIT=]*unum*,NML=*group−name*[,IOSTAT=*rn*][,ERR=*s*][,END=*eof*])

Method of Operation

A namelist−directed **WRITE** statement transfers data from internal storage to external records. It translates the data from internal to external format using the data type of the items in the corresponding **NAMELIST** statement (see Chapter 4, "Specification Statements.") A namelist−directed **READ** or **ACCEPT** statement can read the output of a namelist−directed **WRITE** statement.

Rules for Use

Namelist items are written in the order that referenced **NAMELIST** defines them.

Examples for All Forms of Sequential WRITE

The following statement writes the prompt **enter a filename** to standard output:

```
    write (*,105)
105 format (1x,'enter a filename')
```

The following statement opens the file **%%temp** and writes the record **pair** to the file.

```
     open (unit=10, status='unknown',file="%%temp")
     write (10,1910) pair
1910 format (A)
```

Control Information List — cilist

This section describes the components of the control information list (*cilist*) and the I/O list (*iolist*), which can be specified as elements of the I/O statements described in this chapter.

Table 8−9 summarizes the items that can be specified in a cilist. Each *cilist* specifier shown in the table can appear no more than once in a *cilist*. Note that the keywords **UNIT=** and **FMT=** are optional. Normally, the *cilist* items may be written in any order, but if **UNIT=** or **FMT=** is omitted, the following restrictions apply:

- The keyword **UNIT=** can be omitted if and only if the unit specifier is the first item on the list.
- The keyword **FMT=** can be omitted if and only if the format specifier is the second item in the *cilist* and the first item is a unit specifier in which the keyword **UNIT=** has been omitted.

  A format specifier denotes a formatted I/O operation; default is an unformatted I/O operation. If a record specifier is present, then direct access I/O is denoted; default is sequential access.

Table 8−9 Control Information List Specifiers

Specifier	Purpose
[UNIT=]*u*	Unit or internal file to be acted on.
[NML= *group−name*]	Identifies the *group−name* of a list of items for namelist−directed I/O.
[FMT=]*f*	Formatted or unformatted I/O operations. If formatted, contains format specifiers for data to be read or written.
REC= *rn*	Number of a record accessed in direct−access mode.
KEY [*c*] =*val*	Value of the key field in a record accessed in indexed access mode, where c can be the optional match condition EQ, GT, or GE.

KEYID= *kn*	Key−reference specifier, specifying either the primary key or one of the alternate keys in a record referenced in indexed−access mode.
IOSTAT= *ios*	Name of a variable in which I/O completion status is returned.
ERR= *s*	Label of a statement to which control is transferred if an error occurs.
END= *s*	Label of a statement to which control is transferred if an end−of−file condition (READ only) occurs.

Unit Specifier — UNIT

The form of a unit specifier is

`[UNIT=]`*u*

where *u* is a unit identifier specified as follows:

- A nonnegative integer or noninteger expression specifying the unit.

  A noninteger expression is converted to integer, and the fractional portion, if present, is discarded before use.

- An asterisk specifying a unit that is connected for formatted sequential access (external file identifier only). This denotes the system input unit in a **READ** statement or the system output unit in a **WRITE** statement.

- An identifier that is the name of a character variable, character array, character array element, or substring (internal file identifier only).

An external unit identifier can have the form described in the first or second rule above, except that it cannot be an asterisk in an auxiliary output statement.

An internal file identifier must be specified in the third rule above.

The syntax shows that you can omit the **UNIT=** keyword. If **UNIT=** is omitted, the unit identifier must be first in a control information list. For example, two equivalent **READ** statements are

```
READ(UNIT=5)
READ(5)
```

Format Specifier — FMT

The syntax of a format specifier is

`[FMT=]`*f*

where

f is a format identifier. As shown in the syntax, the keyword **FMT=** can be omitted from the format identifier. If so, the format identifier must be second in a control information list, and the **UNIT=** keyword must also have been omitted.

The legal kinds of format identifiers are

- the statement label of a **FORMAT** statement (the **FORMAT** statement and the format identifier must be in the same program unit)

- an integer variable name assigned to the statement label of a **FORMAT** statement (the **FORMAT** statement and the format identifier must be in the same program unit)
- a character expression (provided it does not contain the concatenation of a dummy argument that has its length specified by an asterisk)
- the name of a character array
- an asterisk that is used to indicate list−directed formatting

Namelist Specifier — NML

The namelist specifier indicates namelist−directed I/O within the **READ** or **WRITE** statement where **NML** is specified. It has the format

`[NML=]`*group−name*

where *group−name* identifies the list in a previously defined **NAMELIST** statement (see Chapter 4, "Specification Statements.")

NML can be omitted when preceded by a unit specifier (unum) without the optional **UNIT** keyword.

Record Specifier — REC

The form of a record specifier is

`REC=`*rn*

where *rn* is an expression that evaluates the record number of the record to be accessed in a direct−access I/O operation. Record numbers must be integers greater than zero.

Key−Field−Value Specifier — KEY

The indexed−access method uses the key−field−value specified in **READ**, **REWRITE**, or other I/O statement. A key field in the record is used as criteria in selecting a record from an indexed file. The key fields for the records in an indexed file are established by the **KEY** specifier used in the **OPEN** statement that created the file.

The key−field−value specifier has the forms shown in Table 8−10

Table 8−10 Forms of the Key−Field−Value Specifier

Specifier	**Basis for Record Selection**
KEY=*kval*	Key−field value *kval*
KEYEQ=*kval*	Key−field value *kval* and the key field are equal
KEYGT=*kval*	Key−field value is greater than the key field
KEYGE=*kval*	Key−field value is greater than or equal to the key field

The following rules apply to *kval*:

- *kval* can be a character or integer expression; if an integer expression, it cannot contain any real or complex values. If the indexed file is formatted, *kval* should always be a character expression.

- The character expression can be an ordinary character string or an array name of type **LOGICAL*1** or **BYTE** containing Hollerith data.
- The character or integer type specified for *kval* must match the type specified for the key field in the record.

Key−of−Reference Specifier — KEYID

The key−of−reference specifier designates, in a **READ**, **REWRITE**, or other I/O statement, the key field in a record to which the key−field−value specifier applies.

The specifier has the following format:

`KEYID=`*n*

where *n* is a number from 0 to the maximum number of keys defined for the records in the indexed file; 0 specifies the primary key, 1 specifies the first alternate key, 2 specifies the second alternate key, and so on. The **KEY** parameter of the **OPEN** statement that created the files creates and establishes the ordering of the primary and alternate keys.

If **KEYID** is not specified, the previous **KEYID** specification in an I/O statement to the same I/O unit is used. The default for **KEYID** is zero (0) if it is not specified for the first I/O statement.

Input/Output Status Specifier — ios

An I/O status specifier has the form

`IOSTAT=`*ios*

where *ios* is a status variable indicating an integer variable or an integer array element. Execution of an I/O statement containing this specifier causes *ios* to become defined with one of the following values:

- Zero if neither an error condition nor an end−of−file condition is encountered by the processor, indicating a successful operation
- Positive integer if an error condition occurred
- Negative integer if an end−of−file condition is encountered without an error condition

  For details about **IOSTAT**, refer to the *perror*(3F) and *intro*(2) manual pages.

Error Specifier — ERR

An error specifier has the following form

`ERR=`*s*

where *s* is an error return label of an executable statement that appears in the same program unit as the error specifier.

If an error condition occurs during execution of an I/O statement with an error specifier, execution of the statement is terminated and the file position becomes indeterminate. If the statement contains an I/O status specifier, the status variable *ios* becomes defined with a processor−dependent positive integer. Execution then continues at the statement labeled *s*.

End−of−File Specifier — END

The form of an end−of−file specifier is

```
END=s
```

where *s* is an end−of−file return label of an executable statement that appears in the same program unit as the end−of−file specifier. An end−of−file specifier may only be used on the *cilist* of a **READ** statement.

If an end−of−file condition is encountered during the execution of a **READ** statement containing an end−of−file specifier and no error occurs, execution of the **READ** statement terminates. If the **READ** statement contains an I/O status specifier, the I/O status variable *ios* becomes defined with a processor−dependent negative integer. Execution then continues at the statement labeled *s*.

Input/Output List — iolist

This section describes the components of I/O list (*iolist*), which can be specified as elements of the I/O statements described in this chapter.

An input/output list specifies the memory locations of the data to be transferred by the I/O statements **READ**, **WRITE**, and **PRINT**.

If an array name is given as an I/O list item, the elements in the array are treated as though each element were explicitly specified in the I/O list in storage order. Note that the name of an assumed−size dummy array (that is, an array declared with an * for an upper bound) must not appear as an I/O list item.

Input List

An input list item can be one of the following:

- Variable name.
- Array element name.
- Substring name.
- Array name.
- Implied **DO** list containing any of the above and other implied−**DO** lists.
- An aggregate reference (a structured data item as defined by a **RECORD** and **STRUCTURE** statement). An aggregate reference can be used only in unformatted input statements. When an aggregate name appears in an *iolist*, only one record is read regardless of how many aggregates or other list items are present.

Examples of input lists are

```
READ(5,3000,END=2000)X,Y(J,K+3),C(2:4)
READ(JFILE,REC=KNUM,ERR=1200)M,SLIST(M,3),cilist
```

Output List

An output list item can be one of the following:

- Variable name.
- Array element name.
- Substring name.
- Array name.
- Any expression, except a character expression involving concatenation of an operand with a length specification of asterisk (*), unless the operand is the symbolic name of a constant.
- An implied **DO** list containing any of the above and other implied **DO** lists.
- An aggregate reference (a structured data item as defined by a **RECORD** and **STRUCTURE** statement). An aggregate reference can be used only in unformatted output statements. When an aggregate name appears in an *iolist*, only one record is written regardless of how many aggregates or other list items are present.

Note that a constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear in an output list but not in an input list.

An example of an output list is

```
WRITE(5,200,ERR=10)'ANSWER IS',N,SQRT(X)+1.23
```

Implied−DO Lists

An implied-**DO** list is a specification that follows the I/O list (*iolist*) in an I/O statement. The list permits the iteration of the statement as though it were contained within a **DO** loop. An implied-**DO** list has the form:

`(`*iolist*`,i=`*e1*`,`*e2*`[,`*e3*`])`

where

iolist	is one or more valid names of the data to be acted on.
i	is an iteration count.
e1, *e2*, and *e3*	are control parameters. See the description of the **DO** statement in Chapter 6, "Control Statements," for a description of *i*, *e1*, *e2*, and *e3*.

The control variable i must not appear as an input list item in *iolist*. The list items in *iolist* are specified once for each iteration of the implied **DO** list with the appropriate substitution of values for each occurrence of the control variable i. When an I/O error occurs within the implied **DO** loop, the value of the control variable i is undefined.

Example

The following statements write **Hello World** to standard output 100 times:

```
write (*,111) ('Hello World',i=1,100)
111 format (1x,A)
end
```

Data Transfer Rules

Data are transferred between records and items specified by the I/O list. The list items are processed in the order in which they appear in the list.

The following restrictions apply to data transfer operations:

- An input list item must not contain any portion of the established format specification.
- If an internal file has been specified, an I/O list item must not be in the file or associated with the file.
- Each output list item must be defined before the transfer of that item.
- All values needed to determine which entities are specified by an I/O list item are determined at the beginning of the processing of that item.

The following sections discuss the rules specific to unformatted and formatted I/O.

Unformatted Input/Output

The execution of an unformatted I/O statement transfers data without editing between the current record and the items specified in the I/O list. Exactly one record is either read or written.

For an unformatted input statement, the record must contain at least as many values as the number of values required by the input list. The data types of the values in the record must agree with the types of the corresponding items in the input list. Character data from an input record must have the same length attribute as the corresponding item in the input list.

The following conventions apply to the execution of an unformatted output statement:

- For direct access, the output list must not specify more values than can fit into a record. If the values specified by the output list do not fill the record, the remainder of the record is filled with zeros.
- For sequential access, the output list defines the size of the output record.

Fortran 77 allows unformatted data transfer only for external files and prohibits it for files connected for formatted I/O.

Formatted Input/Output

The execution of a formatted I/O statement transfers data with editing between the items specified by the I/O list and the file. The current record and possibly additional records are read or written.

Each execution of a **READ** statement causes at least one record to be read, and the input list determines the amount of data to be transferred from the record. The position and form of that data are established by the corresponding format specification.

In a formatted output operation, each execution of the **WRITE** or **PRINT** statement causes at least one record to be written. The amount of data written to the specified unit is determined both by the output list and the format specification.

When a repeatable edit descriptor in a format specification is encountered, a check is made for the existence of a corresponding item in the I/O list. If there is such an item, it transmits appropriately edited information between the item and the record, and then format control proceeds. If there is no corresponding item, format control terminates. Chapter 9, "Format Specification," explains formatted I/O in detail.

Chapter 9

Format Specification

This chapter contains the following subsections:

- "FORMAT Statement"
- "Field and Edit Descriptors"
- "Field Descriptor Reference"
- "Edit Descriptor Reference"
- "Complex Data Editing"
- "Interaction Between I/O List and Format"
- "List−Directed Formatting"

A format specification provides explicit editing information to the processor on the structure of a formatted data record. It is used with formatted I/O statements to allow conversion and data editing under program control. An asterisk (*) used as a format identifier in an I/O statement specifies list−directed formatting.

A format specification may be defined in a **FORMAT** statement or through the use of arrays, variables, or expressions of type character. During input, field descriptors specify the external data fields and establish correspondence between a data field and an input list item. During output, field descriptors are used to describe how internal data is to be recorded on an external medium and to define a correspondence between an output list item and an external data field.

This section describes the **FORMAT** statement, field descriptors, edit descriptors, and list−directed formatting. It also contains a discussion of carriage−control characters for vertical control in printing formatted records.

As extensions to Fortran 77, the compiler supports additional processor−dependent capabilities, which are described in the *Fortran 77 Programmer's Guide*.

Format specifications can be given in two ways: in **FORMAT** statements or as values of character arrays, character variables, and other character expressions.

Format Stored as a Character Entity

In a formatted input or output statement, the format identifier can be a character entity, provided its value has the syntax of a format specification, as detailed below, on execution. This capability allows a character format specification to be read in during program execution.

When the format identifier is a character array name, the format specification is a concatenation of all the elements in the array. When the format identifier is a character array element name, the format specification is only that element of the array. Therefore, format specifications read through a character array name can fill the whole array, while those read through a character array element name must fit in a single element of that array.

FORMAT Statement

The **FORMAT** statement is a non–executable statement that defines a format specification. It has the following syntax:

xx FORMAT *fs*

where

xx is a statement number that is used as an identifier in a **READ**, **WRITE**, **PRINT**, or **ASSIGN**(label) statement.

fs is a format specification (described in ''Format Specification'').

Format Specification

The syntax of a format specification fs is

([*flist*])

where *flist* is a list of format specifiers of one of the following forms, separated by commas:

[*r*]*fd*
ed
[*r*]*fs*

where

r is a positive integer specifying the repeat count for the field descriptor or group of field descriptors. If *r* is omitted, the repeat count is assumed to be 1.

fd is a repeatable edit descriptor or a field descriptor.

ed is a nonrepeatable edit descriptor.

fs is a format group and has the same form as a complete format specification except the *flist* must be non–empty (it must contain at least one format specifier).

The comma used to separate the format specifiers in *flist* can be omitted as follows:

- Between a **P** edit descriptor and immediately following an **F**, **E**, **D**, or **G** edit descriptor (see ''P Edit Descriptor'').
- Before or after a slash edit descriptor (see ''Slash Editing'').
- Before or after a colon edit descriptor (see ''Colon Descriptor'').

Descriptors

Some descriptors can be repeated, others cannot. The repeatable descriptors are

I*w[.m]*	**Z***w[.m]*	**E***w.d[***E***e]*	**G***w.d[***E***e]*	**A***[w]*
O*w[.m]*	**F***w.d*	**D***w.d*	**L***w*	**Q**

where

w and *e*	are nonzero, unsigned integer constants.
d and *m*	are unsigned integer constants.

These descriptors are described in the respective section.

The nonrepeatable descriptors are

/	*k***P**	**TR***c*	**SS**	*n***H***h*…	**$**
:	**T***c*	**S**	**BN**	'*h*… '	
*n***X**	**TL***c*	**SP**	**BZ**	"*h*… "	

where

n and *c*	are nonzero, unsigned integer constants.
k	is an optionally signed integer constant.
h	is one of the characters capable of representation by the processor.

Format Specifier Usage

Each *field descriptor* corresponds to a particular data type I/O list item:

- Integer field descriptors—**I***w*, **I***w.m*, **O***w*, **Z***w*
- Real, double–precision, and complex field descriptors—**F***w.d*, **E***w.d*, **E***w.d***E***e*, **D***w.d*, **G***w. d*, **G***w.d***E***e*
- Logical field descriptor—**L***w*
- Character and Hollerith field descriptors—**A**, **A***w*

O*w*, and **Z***w* are extensions to Fortran 77.

The terms *r, c, n, d, m, e,* and *w* must all be unsigned integer constants, and, additionally, *r, c, n, e*, and *w* must be nonzero. *k* is an optionally signed integer constant. Descriptions of these list items are given in the sections that describe the individual field descriptors.

The repeat specifier *r* can be used only with the **I**, **O**, **Z**, **F**, **E**, **D**, **G**, **L**, and **A** field descriptors and with format groups.

The *d* is required in the **F**, **E**, **D**, and **G** field descriptors. **E***e* is optional in the **E** and **G** field descriptors and invalid in the others.

Use of named constants anywhere in a format specification is not allowed.

Table 9–1contains an alphabetical summary of the field and edit descriptors.

Table 9–1 Summary of Field and Edit Descriptors

Form	Effect
A[*w*]	Transfers character or Hollerith values
BN	Specifies that embedded and trailing blanks in a numeric input field are to be ignored
BZ	Specifies that embedded and trailing blanks in a numeric input field are

	to be treated as zeros
D*w.d*	Transfers real values (D exponent field indicator)
E*w.d*[E*e*]	Transfers real values (E exponent field indicator)
F*w.d*	Transfers real values
G*w.d*	Transfers real values: on input, acts like F descriptor; on output, acts like E or F descriptor, depending on the magnitude of the value
*n*Hc…c	Transfers values between H edit descriptor and an external 'h…' (output only)
I*w*[.*m*]	Transfers decimal integer values
L*w*	Transfers logical values
O*w*[.*m*]	Transfers octal integer values
kP	Scale factor for F, E, D, and G descriptors
S	Restores the default specification for SP and SS
SP	Writes plus characters (+) for positive values in numeric output fields
SS	Suppresses plus characters (+) for positive values in numeric output fields
Tc	Specifies positional tabulation
TLc	Specifies relative tabulation (left)
TRc	Specifies relative tabulation (right)
nX	Specifies that *n* column positions are to be skipped
Z*w*[.*m*]	Transfers hexadecimal integer values
:	Terminates format control if the I/O list is exhausted
/	Record terminator
$	Specifies suppression of line terminator on output (ignored on input)

Each of the field descriptors described Table 9−1is discussed in detail in the following sections.

Variable Format Expressions

Variable format expressions provide a means for substituting run−time expressions for the field width and other parameters of the field and edit descriptors of the statement. Any expression can be enclosed in angle brackets (<>) and used as an integer constant would be used in the same situation. This facility is not available for anything other than a compile−time **FORMAT** statement.

```
Here is an example that uses a variable format expression:
    program VariableExample
    character*12 greeting
    greeting = 'Good Morning!'
    do 110 I = 1, 12
    write (*,115) (greeting)
 115 format (A<I>)
 110 continue
     end
```

In the above example, the field descriptor for greeting has the format **A***w* where *w* is a variable width specifier **I** (initially set to 1) for the *iolist* item **greeting**. In twelve successive **WRITE** operations, **I** is incremented by 1 to produce the following output:

```
G
Go
Goo
Good
Good
Good M
Good Mo
Good Mor
Good Morn
Good Morni
Good Mornin
Good Morning
```

The following rules apply to variable format expressions:

- Functions calls, references to dummy, and any valid Fortran expression can be specified.
- Non−integer data types are converted to integers before processing.
- The same restrictions on size that apply to any other format specifier also apply to the value of a variable format expression.
- Run−time formats cannot use variable format descriptions.
- If the value of a variable changes during a **READ** or **WRITE** operation, the new value is used the next time it is referenced in an I/O operation.

General Rules for Using FORMAT

Because **FORMAT** allows *exact* specification of input and output format, it is necessarily complex. Some guidelines to its correct usage are outlined below.

- A **FORMAT** statement must always be labeled.
- In a field descriptor such as *r***I***w*[*.m*] or *n***X**, the terms *r*, *w*, and *n* must be unsigned integer constants greater than zero. The term *m* must be an unsigned integer constant whose value is greater than or equal to zero; it cannot be a symbolic name of a constant. The repeat count *r* can be omitted.
- In a field descriptor such as **F***w.d*, the term *d* must be an unsigned integer constant. *d* must be specified with **F**, **E**, **D**, and **G** field descriptors, even if *d* is zero. The decimal point is also required. Both *w* and *d* must be specified. In a field descriptor such as **E***w.d***E***e*, the term *e* must also be an unsigned, nonzero integer constant.
- In an **H** edit descriptor such as *n***H***c1 c2.\ .\ .c subn*, exactly *n* characters must follow the **H**. Any character in the processor character set can be used in this edit descriptor.
- In a scale factor of the form *k***P**, *k* must be an optionally signed integer constant. The scale factor affects the **F**, **E**, **D**, and **G** field descriptors only. Once a scale factor is specified, it applies to all subsequent real field descriptors in that format specification until another scale factor appears; *k* must

be zero (**0P**) to reinstate a scale factor of zero. A scale factor of **0P** is initially in effect at the start of execution of each I/O statement.

- No repeat count *r* is permitted in **BN**, **BZ**, **S**, **SS**, **SP**, **H**, **X**, **T**, **TR**, **TL**, **:**, **/**, **$**, **'** descriptors unless these descriptors are enclosed in parentheses and treated as a format group.
- If the associated I/O statement contains an I/O list, the format specification must contain at least one **I**, **O**, **Z**, **F**, **E**, **D**, **G**, **L**, or **A** field descriptor.
- A format specification in a character variable, character substring reference, character array element, character array, or character expression must be constructed in the same way as a format specification in a **FORMAT** statement, including the opening and closing parentheses. Leading blanks are permitted, and any characters following the closing parenthesis are ignored.
- The first character in an output record generally contains carriage control information. See ''Output Rules Summary'' and ''Carriage Control''.
- A slash (**/**) is both a format specifier list separator and a record terminator. See ''Slash Editing'' for details.
- During data transfers, the format specification is scanned from left to right. A repeat count, *r*, in front of a field descriptor or group of field descriptors enclosed in parentheses causes that descriptor or group of descriptors to be repeated *r** before left to right scanning is continued.

Input Rules Summary

Guidelines that apply specifically to input are

- A minus sign (−) must precede a negative value in an external field; a plus sign (+) is optional before a positive value.
- An external field under **I** field descriptor control must be in the form of an optionally signed integer constant, except that leading blanks are ignored and the interpretation of embedded or trailing blanks is determined by a combination of any **BLANK =** specifier and any **BN** or **BZ** blank control that is currently in effect (see ''BN Edit Descriptor'' and ''BZ Edit Descriptor'').
- An external field under **F**, **E**, **D**, or **G** field descriptor control must be in the form of an optionally signed integer constant or a real constant, except that leading blanks are ignored and the interpretation of embedded or trailing blanks is determined by a combination of any **BLANK =** specifier and any **BN** or **BZ** blank control that is currently in effect (see ''BN Edit Descriptor'' and ''BZ Edit Descriptor'').
- If an external field contains a decimal point, the actual size of the fractional part of the field, as indicated by that decimal point, overrides the *d* specification of the corresponding real field descriptor.
- If an external field contains an exponent, the current scale factor *k***P** descriptor has no effect for the conversion of that field.
- The format specification together with the input list must not attempt to read beyond the end of a

record.

Output Rules Summary

Guidelines that apply specifically to output are

- A format specification cannot specify more output characters than the value in the record length specifier (see ‘‘OPEN’’ of Chapter 8 for details). For example, a line printer record might be limited to no more than 133 characters, including the carriage−control character.
- The field−width specification, *w*, and exponent digits, *e*, must be large enough to accommodate all characters that the data transfer can generate, including an algebraic sign, decimal point, and exponent. For example, the field width specification in an **E** field descriptor should be large enough to contain $d + 6$ characters or $d + e + 4$ characters. The first character of a record of a file intended to be printed is typically used for carriage control; it is not printed. The first character of such a record should be a space, 0, 1, or +. (See ‘‘Carriage Control’’.)

Field and Edit Descriptors

The format specifiers in a format specification consist of field, or repeatable, descriptors and other nonrepeatable edit descriptors.

On input, the field descriptors specify what type of data items are to be expected in the external field so that data item values can be properly transferred to their internal (processor) representations.

On output, the field descriptors specify what type of data items should be written to the external field.

On input and output, the other nonrepeatable edit descriptors position the processor pointer in the external field so that data items will be transferred properly. For instance, edit descriptors can specify that lines or positions in the external field be skipped or that data items can be repeatedly read (on input) or written (on output).

Field Descriptor Reference

This section contains an overview of the numeric field descriptors **I**, **O**, **Z**, **F**, **E**, **D**, and **G**. It also describes the **P** edit descriptor and the **L**, **A**, **H**, **Q**, and character edit descriptors.

Numeric Field Descriptors

The **I**, **O**, **Z**, **F**, **E**, **D**, and **G** field descriptors are used for numeric editing. This section also describes the **P** edit descriptor, which is a scale factor, that alters the effect of **F**, **E**, **D**, and **G** field descriptors.

Unless otherwise indicated, the following rules apply:

- On input, these numeric field descriptors ignore leading blanks in the external field. If a **BZ** edit descriptor is in effect, embedded and trailing blanks are treated as zeros; otherwise, a **BN** edit descriptor is in effect, and all embedded and trailing blanks are ignored. Either **BZ** or **BN** is initially in effect at the beginning of the input statement depending on the **BLANK =** specified (see ‘‘OPEN’’). The default is **BN**.

- A plus sign (+) is produced on output only if **SP** is in effect; however, a minus sign (-) is produced where applicable. When computing the field width for numeric descriptors, one character should be allowed for the sign, whether it is produced or not.
- For input with **F**, **E**, **D**, and **G** descriptors, a decimal point in the input field overrides the **D** specification, and an explicit exponent in the input field overrides the current scale factor.
- For output, fields are right justified. If the field width is too small to represent all required characters, asterisks are produced. This includes significant digits, sign, decimal point, and exponent.

Default Field Descriptor Parameters

You can optionally specify a field−width value (*w*, *d*, and *e*) for the **I**, **O**, **Z**, **L**, **F**, **E**, **D**, **G**, and **A** field descriptors. If you do not specify a value, the default values shown in Table 9−2apply. The length of the I/O variable determines the length *n* for the **A** field descriptor.

Table 9−2 Default Field Descriptors

Descriptor	Field Type	w	d	e
I,O,Z	BYTE	7		
I,O,Z	INTEGER*2, LOGICAL*2	7		
I,O,Z	INTEGER*4, LOGICAL*4	12		
O,Z	REAL*4	12		
O,Z	REAL*8	23		
O,Z	REAL*16	44		
L	LOGICAL	2		
F,E,G,D	REAL, COMPLEX*8	15		
F,E,G,D	REAL*8, COMPLEX*16	25	16	2
F,E,G,D	REAL*16	42	33	3
A	LOGICAL*1	1		
A	LOGICAL*2, INTEGER*2	2		
A	LOGICAL*4, INTEGER*4	4		
A	REAL*4, COMPLEX*8	4		
A	REAL*8, COMPLEX*16	8		
A	REAL*26	16		
A	CHARACTER*n	n		

I Field Descriptor

The **I** field descriptor is used for conversion between an internal integer data item and an external decimal integer. It has the form

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

where

w is a nonzero, unsigned integer constant denoting the size of the external field, including blanks and a sign, if necessary. A minus sign (−) is always printed on output

	if the number is negative. If the number is positive, a plus sign (+) is printed only if SP is in effect.
m	is an unsigned integer constant denoting the minimum number of digits required on output. *m* is ignored on input. The value of *m* must not exceed *w*; if *m* is omitted, a value of 1 is assumed.

In an input statement, the **I** field descriptor reads a field of *w* characters from the record, interprets it as an integer constant, and assigns the integer value to the corresponding I/O list item. The corresponding I/O list element must be of the **INTEGER** or **LOGICAL** data type. The external data must have the form of an integer constant; it must not contain a decimal point or exponent.

A **LOGICAL** data type is displayed as either the value 0 (false) or 1 (true).

If the first nonblank character of the external field is a minus sign, the field is treated as a negative value. If the first nonblank character is a plus sign, or if no sign appears in the field, the field is treated as a positive value. An all−blank field is treated as a value of zero.

Table 9−3contains input examples.

Table 9−3I Field Input Examples

Format	External Field	Internal Value
i4	3244	3244
i3	−15	−15
i9	213	213

In an output statement, the **I** field descriptor constructs an integer constant representing the value of the corresponding I/O list item and writes it to the right−justified record in an external field w characters long. If the value does not fill the field, leading blanks are inserted; if the value exceeds the field width, the entire field is filled with asterisks. If the value of the list item is negative, the field will have a minus sign as its left most, nonblank character. The term *w* must therefore be large enough to provide for a minus sign, when necessary. If *m* is present, the external field consists of at least *m* digits, with leading zeros, if necessary.

If *m* is zero, and the internal representation is zero, the external field is filled with blanks.

Table 9−4contains output examples.

Table 9−4I Field Output Examples

Format	Internal Value	External Field
I3	311	311
i4	−311	−311
i5	417	417
i2	7782	**
i3	−213	***
i4.2	1	01
i4.4	1	0001
i4.0	1	

O Field Descriptor

The **O** field descriptor transfers data values and converts them to octal form. It has the form

O*w*[*m*]

where

w is a nonzero, unsigned integer constant denoting the size of the external field, including blanks and a sign, if necessary. A minus sign (−) is always printed on output if the number is negative. If the number is positive, a plus sign (+) is printed only if **SP** is in effect.

m is an unsigned integer constant denoting the minimum number of digits required on output. *m* is ignored on input. The value of *m* must not exceed *w*; if *m* is omitted, a value of 1 is assumed.

This repeatable descriptor interprets and assigns data in the same way as the **I** field descriptor, except that the external field represents an *octal* number constructed with the digits 0 through 7. On input, if **BZ** is in effect, embedded and trailing blanks in the field are treated as zeros; otherwise, blanks are ignored. On output, **S**, **SP**, and **SS** do not apply.

In an input statement, the field is terminated when a non−octal digit is encountered. Fortran 77 treats embedded and trailing blanks as zeros.

In an input statement, the **O** field descriptor reads *w* characters from the record; the input field must have:

- optional leading blanks
- an optional plus or minus sign
- a sequence of octal digits (0 through 7)

A field that is entirely blank is treated as the value zero.

Table 9−5contains examples of **O** field input values. **BN** is assumed in effect, and internal values are expressed in decimal (base 10).

Table 9−5O Field Input Examples

Format	External Field (INTEGER*4)	Internal Value
o20	−77	−63
o20	1234	668
o20	177777	65535
o20	100000	32768

In an output statement, the **O** field descriptor constructs an octal number representing the unsigned value of the corresponding I/O list element as follows:

- The number is right justified with leading zeros inserted (if necessary). Fortran 77 inserts leading blanks.

- If *w* is insufficient to contain all the digits necessary to represent the unsigned value of the output list item, then the entire field is filled with asterisks.

Table 9–6lists examples of **O** field output.

Table 9–6O Field Output Examples

Format	Internal Value (INTEGER*4)	External Field
o20.2	3	03
o20.2	–1	37777777777
o3	–1	***
o20.2	63	77
O20.2	–2	37777777776

Z Field Descriptor

The **Z** field descriptor transfers data values and converts them to hexadecimal form. It has the form

Z*w*[*m*]

where

w is a nonzero, unsigned integer constant denoting the size of the external field.

m is an unsigned integer constant denoting the minimum number of digits required on output. *m* is ignored on input. The value of *m* must not exceed *w*; if *m* is omitted, a value of 1 is assumed.

This repeatable descriptor interprets and assigns data in the same way as the **I** field descriptor, except that the external field represents a hexadecimal number constructed with the digits 0 through 9 and the letters A through F. On output, the output list item is interpreted as an unsigned integer value.

In an input statement, the **O** field descriptor reads *w* characters from the record. After embedded and trailing blanks are converted to zeros or ignored, as applicable, the input field must have

- optional leading blanks
- an optional plus or minus sign
- a sequence of hexadecimal digits (0 through 9, A through F)

A field that is entirely blank is given a value of zero.

Table 9–7lists examples of **Z** field input. **BN** is assumed in effect, and internal values are expressed in decimal (base 10).

Table 9–7Z Field Input Examples

Format	External Field (INTEGER*4)	Internal Value
Z10	–ff	–255
z10	1234	4660
z10	ffff	65535

z10	8000	32768

Table 9−8lists examples of **Z** field output.

Table 9−8 Z Field Output Examples

Format	Internal Value (INTEGER*4)	External Field
z10.2	3	" 03"
z10.2	−1	" ffffffff"
z10.2	63	" 3f"
z10.2	−2	" fffffffe"

F Field Descriptor

The **F** field descriptor transfers real values. It has the form

F*w***.***d*

where

w is a nonzero, unsigned integer constant denoting field width.

d is an unsigned integer constant denoting the number of digits in the fractional part.

The corresponding I/O list element must be of type **REAL**, **DOUBLEPRECISION**, or **COMPLEX**.

In an input statement, the **F** field descriptor reads a field of *w* characters from the record and, after appropriate editing of leading, trailing, and embedded blanks, interprets it as an integer or a real constant. It then assigns the real value to the corresponding I/O list element. (Refer to Chapter 2, "Constants and Data Structures," for more information.) If the external field contains an exponent, the letter E can be omitted as long as the value of the exponent is a signed integer. If the first nonblank character of the external field is a minus sign, the field is treated as a negative value. If the first nonblank character is a plus sign, or if no sign appears in the field, the field is treated as a positive value. An all−blank field is given a value of zero.

If the field contains neither a decimal point nor an exponent, it is treated as a real number in which the right most *d* digits are to the right of the decimal point, with leading zeros assumed if necessary. If the field contains an explicit decimal point, the location of that decimal point overrides the location specified by the value of *d* in the field descriptor. If the field contains a real exponent, the effect of any associated scale factor *k***P** (see Scale Factor) is suppressed, and the real exponent is used to establish the magnitude of the value in the input field before it is assigned to the list element.

Table 9−9provides examples of **F** field input.

Table 9−9F Field Input Examples

Format	External Field	Internal Value
f8.5	123456789	0.12345678E+03
f8.5	−1234.567	−0.123456E+04
f8.5	12.34e+2	0.1234E+02
F5.2	1234567.89	0.12345E+03

In an output statement, the **F** field descriptor constructs a basic real constant representing the value of the corresponding I/O list element, rounded to *d* decimal positions, and writes it to the record right−justified in an external field *w* characters long.

The term *w* must be large enough to include:

- a minus sign for a negative value or a plus sign (when **SP** is in effect) for a positive value
- the decimal point
- *d* digits to the right of the decimal

If *w* is insufficiently large, the entire field width is filled with asterisks. Therefore, *w* must be > *d* + 2.

Table 9−10provides examples of **F** field output.

Table 9−10F Field Output Examples

Format	Internal Value	External Field
F8.5	.12345678E+01	1.23457
f9.3	.87654321E+04	8765.432
F2.1	.2531E+02	**
f10.4	.1234567E+02	12.3457
f5.2	.123456E+03	******
F5.2	−.4E+00	−0.40

E Field Descriptor

The **E** field descriptor transfers real values in exponential form. It has the form

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

where

w	is a nonzero, unsigned integer constant denoting field width.
d	is an unsigned integer constant denoting the number of digits in the fractional part.
e	is a nonzero, unsigned integer constant denoting the number of digits in the exponent part. The *e* has no effect on input.

The corresponding I/O list element must be of **REAL**, **DOUBLEPRECISION**, or **COMPLEX** data type.

In an input statement, the **E** field descriptor interprets and assigns data in exactly the same way as the**F** field descriptor.

Table 9−11provides examples of **E** field input.

Table 9−11E Field Output Examples

Format	External Field	Internal Value
e9.3	" 654321E3"	.654321E+06
e12.4	" 1234.56E−6"	.123456E−02
e15.3	"12.3456789"	.123456789E+02

e12.5	"123.4567d+10"	.1234567E+13

In Table 9−11, the**E** field descriptor treats the **D** exponent field indicator the same as an **E** exponent indicator.

In an output statement, the **E** field descriptor constructs a real constant representing the value of the corresponding I/O list element, rounded to *d* decimal digits, and writes it to the right−justified record in an external field *w* characters long. If the value does not fill the field, leading spaces are inserted; if the value exceeds the field width, the entire field is filled with asterisks.

When an **E** field descriptor is used, data output is transferred in a standard form. This form consists of

- minus sign for a negative value or a plus sign (when **SP** is in effect) for a positive value
- digits to the left of the decimal point, if any, or an optional zero
- decimal point
- *d* digits to the right of the decimal point
- an *e* + 2−character exponent or a 4−character exponent

The exponent has one of the following forms:

E*w.d*	**E** + *nn* or **E** -*nn* if the value of the exponent is in the range of -99 to +99
E*w.d*	+*nnn* or -*nnn* if the value of the exponent is <= -99 or <= +99
E*w.d***E***e*	**E** + *n1 n2…n* sub *e* or **E**- *n1 n2…n sub e*, where *n1 n2… ne* is the magnitude of the exponent with leading zeros, if necessary.

The exponent field−width specification is optional; if it is omitted, the exponent part is as shown above. If the exponent value is too large to be output with the given value *e* as shown in the third form above, the entire field is filled with asterisks.

The term *w* must be large enough to include

- A minus sign when necessary (plus signs when **SP** is in effect)
- All significant digits to the left of the decimal point
- A decimal point
- *d* digits to the right of the decimal point
- The exponent

Given these limitations and assuming a **P** edit descriptor is in effect, *w* is \xb3 *d* + 7, or \xb3 *d* + *e* + 5 if *e* is present.

Table 9−12provides examples of **E** field output.

Table 9−12E Field Output Examples

Format	**Internal Value**	**External Field**

E9.2	.987654321E+06	" .99E+06"
e12.5	.987654321E+06	" .98765E+06"
e12.3	.69E-5	" .690E-05"
e10.3	-.5555E+00	" –.556E+00"
e5.3	.7214E+02	"*****"
e14.5E4	-.1001E+01	" -.10010E+0001"
e14.3E6	.123e-06	" .123E-000003"

D Field Descriptor

The **D** field descriptor transfers real values in exponential form. It has the form

D*w*.*d*

where

w is a nonzero, unsigned integer constant denoting field width.

d is an unsigned integer constant denoting the number of digits in the fractional part.

The corresponding I/O list element must be of **REAL**, **DOUBLEPRECISION**, or **COMPLEX** data type.

In an input statement, the **D** field descriptor interprets and assigns data in exactly the same way as the**F** field descriptor.

Table 9–13provides examples of **D** field input.

Table 9–13D Field Input Examples

Format	External Field	Internal Value
d10.2	"12345 "	.12345E+03
d10.2	" 123.45"	.12345E+03
d15.3	"123.4567891D+04"	.1234567891E+07

In an output statement, the **D** field descriptor is the same as the **E** field descriptor, except the **D** exponent field indicator replaces the **E** indicator.

Table 9–14provides examples of **D** field output.

Table 9–14D Field Output Examples

Format	Internal Value	External Field
d14.3	123d – 04	" .123D – 04"
d23.12	123456789123d + 04	" .123456789123D + 04"
d9.5	14D + 01	"*********"

G Field Descriptor

A **G** field descriptor is used for the conversion and editing of real data when the magnitude of the data is unknown. On output, the **G** field descriptor produces a field as do the **F** or **E** field descriptors, depending on the value. On input, the **G** field descriptor interprets and assigns data in exactly the same way as the**F** field descriptor. It has the form

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

where

w is a nonzero, unsigned integer constant denoting field width.

d is an unsigned integer constant denoting the number of digits in the basic value part.

e is a nonzero, unsigned integer constant denoting the number of digits in the exponent part.

The corresponding I/O list element must be of **REAL**, **DOUBLEPRECISION**, or **COMPLEX** data type.

In an input statement, the **G** field descriptor interprets and assigns data in exactly the same way as the **F** field descriptor.

In an output statement, the **G** field descriptor constructs a real constant representing the value of the corresponding I/O list element rounded to *d* decimal digits and writes it to the right−justified record in an external field *w* characters long. The form in which the value is written is a function of the magnitude of the value *m*, as described in Table 9−1 In the table, *n* is 4 if **E***e* was omitted from the **G** field descriptor; otherwise *n* is *e* + 2.

Table 9−15illustrates the effect of data magnitude on **G** format conventions.

Table 9−15 Effect of Data Magnitude on G Format Conventions

Data Magnitude	Effective Format
m < 0.1	E*w*.*d*[E*e*]
0.1 \xb2 m < 1.0	F(*w*−*n*).*d*, *n* (’")
1.0 \xb2 m < 10.0	F(*w*−*n*).(*d*−1) (’’)
10 d−2 \xb2 m < 10d−1	F(*w*−*n*).1,*n* (’’)
10 d−1 \xb2 m < 10d	F(*w*−*n*).0*n* (’’)
m \xb3 10d	E*w*.*d*[E*e*]

The term *w* must be large enough to include

- A minus sign for a negative value or a plus sign (when **SP** is in effect) for a positive value
- A decimal point
- *d* digits in the basic value part
- Either a 4−character or *e* + 2−character exponent part

Given these limitations, *w* must therefore be \xb3 *d* + 7 or \xb3 *d* + *e* + 5.

Table 9−16provides examples of **G** field output.

Table 9−16G Field Output Examples

Format	Internal Value	External Field
g13.6	.1234567E−01	" .1234567E−01"
g13.6	−.12345678E00	" −.123457 "

g13.6	.123456789E+01	" 1.23457 "
g13.6	.1234567890E+02	" 12.3457 "
g13.6	.12345678901E+03	" 123.457 "
g13.6	−.123456789012E+04	" −1234.57 "
g13.6	.1234567890123E+05	" 12345.7 "
g13.6	.12345678901234E+06	" 123457. "
g13.6	−.123456789012345E+07	" −.123457E+07"

For comparison, the examples in Table 9−17use the same values with an equivalent **F** field descriptor.

Table 9−17 Field Comparison Examples

Format	Internal Value	External Field
f13.6	.1234567E−01	" .012346"
f13.6	−.12345678E00	" −.123457"
f13.6	.123456789E+01	" 1.234568"
f13.6	.1234567890E+02	" 12.345679"
f13.6	.12345678901E+03	" 123.456789"
f13.6	−.123456789012E+04	" −1234.567890"
f13.6	.1234567890123E+05	" 12345.678901"
f13.6	.12345678901234E+06	"123456.789012"
F13.6	−.123456789012345E+07	"*************"

P Edit Descriptor

The **P** edit descriptor specifies a *scale factor* and has the form

*k*P

where *k* is an optionally signed integer constant called the *scale factor.*

A **P** edit descriptor can appear anywhere in a format specification but must precede the first field descriptor that is to be associated with it. For example

*k*PF*w.d*   *k*PE*w.d*   *k*PD *w.d*   *k*PG*w.d*

The value of *k* must not be greater than *d* + 1, where *d* is the number of digits in the **E***w.d*, **D***w.d*, or **G***w.d* output fields.

Scale Factor

The scale factor, *k*, determines the appropriate editing as follows:

- For input with **F**, **E**, **D**, and **G** editing (provided there is no exponent in the field) and **F** output editing, the magnitude represented by the external field equals the magnitude of the internal value multiplied by 10k.
- For input with **F**, **E**, **D**, and **G** editing containing a real exponent, the scale factor has no effect.
- For output with **E** and **D** editing, the basic value part is multiplied by 10k and the real exponent is reduced by *k*.

- For output with **G** editing, the scale factor has no effect unless the data to be edited is outside the range that permits **F** editing. If the use of **E** editing is required, the effect of the scale factor is the same as **E** output editing. (See Real Type in Chapter 2.)

On input, if no exponent is given, the scale factor in any of the above field descriptors multiplies the data by $10^{-k}$and assigns it to the corresponding I/O list element. For example, a 2**P** scale factor multiplies an input value by .01. A **–2P** scale factor multiplies an input value by 100. However, if the external field contains an explicit exponent, the scale factor has no effect. Table 9–18gives examples of scale factors.

Table 9–18 Scale Factor Examples

Format	External Field	Internal Value
3pe10.5	" 37.614"	.37614E–01
3pe10.5	" 37.614E2"	.37614E+04
–3pe10.5	" 37.614"	.37614e+05

On output, the effect of the scale factor depends on the type of field descriptor associated with it.

For the **F** field descriptor, the value of the I/O list element is multiplied by 10k before transfer to the external record: a positive scale factor moves the decimal point to the right; a negative scale factor moves the decimal point to the left. The value represented is 10k multiplied by the internal value.

For output with the **E** or **D** field descriptor, the basic real constant part of the external field is multiplied by 10k and the exponent is reduced by *k*. The value represented is unchanged. A positive scale factor moves the decimal point to the right and decreases the exponent; a negative scale factor moves the decimal point to the left and increases the exponent. In summation,

$k > 0$	moves the decimal point *k* digits to the right.
$k < 0$	moves the decimal point *k* digits to the left.
$k = 0$	leaves the decimal point unchanged.

Table 9–19provides scale format output examples.

Table 9–19 Scale Format Output Examples

Format	Internal Value	External Field
1pe12.3	–.270139E+03	" 2.701E+0 2"
1pe12.2	–270139E+03	" 2.70E+02"
–1pe12.2	–.270139E+03	" 0.03E+04"

On output, the effect of the scale factor for the **G** field descriptor is suspended if the magnitude of the output data is within the range permitting **F** editing because the **G** field descriptor supplies its own scaling function. The **G** field descriptor functions as an **E** field descriptor if the magnitude of the data value is outside its range. In this case, the scale factor has the same effect as the **E** field descriptor.

On output under **F** field descriptor control, a scale factor actually alters the magnitude of the value represented, multiplying or dividing it by ten. On output, a scale factor under **E**, **D**, or **G** field descriptor control merely alters the form in which the value is represented.

If you do not specify a scale factor with a field descriptor, a scale factor of zero is assumed at the beginning of the execution of the statement. Once a scale factor is specified, it applies to all subsequent **F**, **E**, **D**, and **G** field descriptors in the same format specification, unless another scale factor appears. A scale factor of zero can be reinstated only with an explicit **P** specification.

L Edit Descriptor

The **L** edit descriptor is used for logical data. The specified I/O list item must be of type **LOGICAL**. It has the form

L*w*

where *w* is a nonzero, unsigned integer constant denoting field width.

For input, the field must consist of optional blanks followed by an optional decimal point followed by a **T** (for true) or **F** (for false). The **T** or **F** can be followed by additional characters that have no effect. The logical constants **.TRUE.** and **.FALSE.** are acceptable input forms.

For output, the field consists of *w* - 1 blanks followed by a **T** or an **F**, for true and false, respectively, according to the value of the internal data. Table 9–20 shows **L** field examples.

Table 9–20 L Field Examples

Format	Internal Value	External Field
L5	.TRUE.	" T"
l1	.FALSE.	"F"

The **L** edit descriptor can also be used to process integer data items. All nonzero values are displayed as **.TRUE.** and all zero values as **.FALSE.**.

A Edit Descriptor

The **A** edit descriptor is used for editing character or Hollerith data. It has the form

A[*w*]

where *w* is a nonzero, unsigned integer constant denoting the width, in number of characters, of the external data field. If *w* is omitted, the size of the I/O list item determines the length *w*.

The corresponding I/O list item can be any data type. If it is character data type, character data is transmitted. If it is any other data type, Hollerith data is transmitted.

In an input statement, the **A** edit descriptor reads a field of *w* characters from the record without interpretation and assigns it to the corresponding I/O list item. The maximum number of characters that can be stored depends on the size of the I/O list item. For character I/O list elements, the size is the length of the character variable, character substring reference, or character array element. For numeric and logical I/O list elements, the size depends on the data type, as shown in Table 9–21

Table 9–21 I/O List Element Sizes

I/O List Element	Maximum Number of Characters
LOGICAL*1	1
LOGICAL*2	2

LOGICAL*4	4
INTEGER*2	2
INTEGER*4	4
REAL*4 (REAL)	4
REAL*8 (DOUBLE PRECISION)	8
COMPLEX*8 (COMPLEX)	8
COMPLEX*16 (DOUBLE COMPLEX)	16

If *w* is greater than the maximum number of characters that can be stored in the corresponding I/O list item, only the right most characters of the field are assigned to that element. The left most excess characters are ignored. If *w* is less than the number of characters that can be stored, *w* characters are assigned to the list item and left justified, and trailing blanks are added to fill it to its maximum size.

Input Example

Table 9−22lists **A** field input examples.

Table 9−22 A Field Input Examples

Format	External Field	Internal Value	Representation
A6	"FACE #"	"#"	(CHARACTER*1)
A6	"FACE #"	"E #"	(CHARACTER*3)
A6	"FACE #"	"FACE #"	(CHARACTER*6)
A6	"FACE #"	"FACE # "	(CHARACTER*8)
A6	"FACE #"	"#"	(LOGICAL*1)
A6	"FACE #"	"#"	(INTEGER*2)
A6	"FACE #"	"CE #"	(REAL*4)
A6	"FACE #"	"FACE # "	(REAL*8)

In an output statement, the **A** field descriptor writes the contents of the corresponding I/O list item to the record as an external field *w* characters long. If *w* is greater than the list item size, the data appears in the field, right justified, with leading blanks. If *w* is less than the list element, only the left most *w* characters from the I/O list item are transferred.

Table 9−23lists **A** field output examples.

Table 9−23A Field Output Examples

Format	Internal Value	External Field
A6	"GREEK"	" GREEK"
A6	"FRENCH"	"FRENCH"
A6	"PORTUGUESE"	"PORTUG"

If you omit *w* in an **A** field descriptor, a default value is supplied based on the data type of the I/O list item. If it is character type, the default value is the length of the I/O list element. If it is numeric or logical data type, the default value is the maximum number of characters that can be stored in a variable of that data type as described for input.

Repeat Counts

The **I**, **O**, **Z**, **F**, **E**, **D**, **G**, **L**, and **A** field descriptors can be applied to a number of successive I/O list items by preceding the field descriptor with an unsigned integer constant, called the repeat count. For example, 4**F**5.2 is equivalent to **F**5.2, **F**5.2, **F**5.2, **F**5.2.

Enclosing a group of field descriptors in parentheses, and preceding the enclosed group with a repeat count, repeats the entire group. Thus, 2(I6,**F**8.4) is equivalent to **I**6,**F**8.4,**I**6,F8.4.

H Field Descriptor

The **H** field descriptor is used for output of character literal data. It has the form:

*n*H*xxx*… *x*

where

n	is an unsigned integer constant denoting the number of characters that comprise the character literal.
x	comprises the character literal and consists of *n* characters, including blanks.

In an output statement, the **H** field descriptor writes the *n* characters following the letter **H** from the field descriptor to the record as an external field *n* characters long. The **H** field descriptor does not correspond to an output list item.

Table 9−24lists examples of **H** edit description output.

Table 9−24H Edit Description Output Examples

Specification	External Field
6HAb CdE	Ab CdE
1H9	9
4H'a2'	'a2'

An **H** field descriptor must not be encountered by a **READ** statement.

Character Edit Descriptor

A character edit descriptor has one of the following forms:

'X1 X2 … Xn'
X1 X2 … Xn

where *X1 X2 … Xn* are members of the Fortran character set forming a valid character literal. The width of the output field is the number of characters contained in the character literal, excluding the enclosing apostrophes or quotation marks. The character edit descriptor does not correspond to an output list item. Within a character edit descriptor delimited by apostrophes, an apostrophe is represented by two successive apostrophe characters. Within a character edit descriptor delimited by quotation marks, a quotation mark is represented by two successive quotation mark characters.

Example

Table 9−25lists character edit description examples.

Table 9–25 Character Edit Description Examples

Output Specification	External Field
'sum ='	sum =
.sum =	sum =
.don't	don't
'here''s the answer'	here's the answer
'he said, "yes"'	he said, "yes"
.he said, ""yes""	he said, "yes"

A character edit descriptor must not be encountered by a **READ** statement.

Use of quotation marks as a character edit descriptor is an enhancement to Fortran 77.

Q Edit Descriptor

The **Q** edit descriptor is used to determine the number of characters remaining to be read from the current input record. It has the form

```
Q
```

When a **Q** descriptor is encountered during the execution of an input statement, the corresponding input list item must be type integer. Interpretation of the **Q** edit descriptor causes the input list item to be defined with a value that represents the number of character positions in the formatted record remaining to be read. Therefore, if *c* is the character position within the current record of the next character to be read and the record consists of *len* characters, then the item is defined with the value

n = max (*len* – *c* + 1, 0)

If no characters have yet been read, then *n* = *len*, the length of the record. If all the characters of the record have been read (*c* > *len*), then *n* is zero.

The **Q** edit descriptor must not be encountered during the execution of an output statement.

Input Example

The following is an example of **Q** edit description input:

```
     INTEGER N
     CHARACTER LINE * 80
     READ (5, 100) N, LINE (1:N)
100  FORMAT (Q, A)
```

Edit Descriptor Reference

After each **I**, **O**, **Z**, **F**, **E**, **D**, **G**, **L**, **A**, **H**, or character edit descriptor is processed, the file is positioned after the last character read or written in the current record.

The **X**, **T**, **TL**, and **TR** descriptors specify the position at which the next character will be transmitted to or from the record. They do not change any characters in the record already written or by themselves affect the length of the record.

If characters are transmitted to positions at or after the position specified by a **T**, **TL**, **TR**, or **X** edit descriptor, positions skipped and not previously filled are filled with blanks.

X Edit Descriptor

The **X** edit descriptor specifies a position forward (to the right) of the current position. It is used to skip characters on the external medium for input and output. It has the form

n`X`

where *n* is a nonzero, unsigned integer constant denoting the number of characters to be skipped.

T Edit Descriptor

The **T** edit descriptor specifies an absolute position in an input or output record. It has the form:

`T`*n*

where *n* indicates that the next character transferred to or from the record is the *n*th character of the record.

TL Edit Descriptor

The **TL** edit descriptor specifies a position to the left of the current position. It has the form

`TL`*n*

where *n* indicates that the next character to be transferred from or to the record is the *n*th character to the left of the current character. The value of *n* must be greater than or equal to one.

If *n* is the current character position, then the first character in the record is specified.

TR Edit Descriptor

The **TR** edit descriptor specifies a position to the right of the current position. It has the form

`TR`*n*

where *n* indicates that the next character to be transferred from or to a record is the *n*th character to the right of the current character. The value of *n* must be greater than or equal to one.

BN Edit Descriptor

The **BN** edit descriptor causes the processor to ignore blank characters in a numeric input field and to right justify the remaining characters, as though the blanks that were ignored were leading blanks. It has the form

`BN`

The **BN** descriptor affects only **I**, **O**, **Z**, **F**, **E**, **D**, and **G** editing and then only on input fields.

BZ Edit Descriptor

The **BZ** edit descriptor causes the processor to treat all the embedded and trailing blank characters it encounters within a numeric input field as zeros. It has the form:

```
BZ
```

The **BZ** descriptor affects only **I**, **O**, **Z**, **F**, **E**, **D**, and **G** editing and then only on input fields.

SP Edit Descriptor

The **SP** edit descriptor specifies that a plus sign be inserted in any character position that normally contains an optional plus sign and whose actual value is \xb3 0. It has the form

```
SP
```

The **SP** descriptor affects only **I**, **F**, **E**, **D**, and **G** editing and then only on output fields.

SS Edit Descriptor

The **SS** edit descriptor specifies that a plus sign should *not* be inserted in any character position that normally contains an optional plus sign. It has the form

```
SS
```

The **SS** descriptor affects only **I**, **F**, **E**, **D**, and **G** editing and then only on output fields.

S Edit Descriptor

The **S** edit descriptor resets the option of inserting plus characters (+) in numeric output fields to the processor default. It has the form

```
S
```

The **S** descriptor counters the action of either the **SP** or the **SS** descriptor by restoring to the processor the discretion of producing plus characters (+) on an optional basis. The default is to **SS** processing; the optional plus sign is not inserted when **S** is in effect.

The **S** descriptor affects only **I**, **F**, **E**, **D**, and **G** editing and then only on output fields.

Colon Descriptor

The colon character (:) in a format specification terminates format control if no more items are in the I/O list. The colon descriptor has no effect if I/O list items remain.

$ Edit Descriptor

The $ edit descriptor suppresses the terminal line−mark character at the end of the current output record. It has the form

```
$
```

The $ descriptor is nonrepeatable and is ignored when encountered during input operations.

Output Example

```
       print 100, 'enter a number:'
100    format (1x, a, $)
       read *, x
```

Complex Data Editing

A complex value consists of an ordered pair of real values. If an **F**, **E**, **D**, or **G** field descriptor is encountered, and the next I/O list item is complex, then the descriptor is used to edit the real part of the complex item. The next field descriptor is used to edit the imaginary part.

If an **A** field descriptor is encountered on input or output, and the next I/O list item is complex, then the **A** field descriptor is used to translate Hollerith data to or from the external field and the entire complex list item. The real and imaginary parts together are treated as a single I/O list item.

In an input statement with **F**, **E**, **D**, or **G** field descriptors in effect, the two successive fields are read and assigned to a complex I/O list element as its real and imaginary parts, respectively.

Table 9–26contains examples of complex data editing input.

Table 9–26 Complex Data Editing Input Examples

Format	External Field	Internal Value
f8.5,f8.5	"1234567812345.67"	(.12345678E+03,.1234567E+05)
f9.1,f9.3	"734.432E8123456789"	(.734432E+11,.123456789E+06)

In an output statement with **F**, **E**, **D**, or **G** field descriptors in effect, the two parts of a complex value are transferred under the control of successive field descriptors. The two parts are transferred consecutively, without punctuation or spacing, unless the format specification states otherwise.

Table 9–27contains examples of complex data editing output.

Table 9–27 Complex Data Editing Output Examples

Format	Internal Value	External Field
2f8.5	(.23547188E+01,.3456732E+01)	"2.35472 3.45673"
e9.2,",",e5.3	(.47587222E+05,.56123E+02)	"0.48E+06, *****"

Carriage Control

A formatted record can contain a prescribed carriage–control character as the first character of the record. The carriage–control character determines vertical spacing in printing when the **CARRIAGECONTROL** keyword of the **OPEN** statement is set to **FORTRAN** (as described in ‘‘OPEN’’ of Chapter 8, "Input/Output Statements.") Table 9–28lists the carriage–control characters.

Table 9–28 Carriage–Control Characters

Character	Effect on Spacing
Blank	Single space
0	Double space
1	To first line of next page

+	No vertical spacing
$	Output starts at the beginning of the next line; carriage return at the end of the line is suppressed
ASCII NUL	Overprints with no advance; does not return to the left margin after printing

The carriage−control character is not printed, and the remaining characters, if any, are printed on one line beginning at the left margin. If there are no characters in the record, the vertical spacing is one line and no characters will be printed in that line.

Slash Editing

A slash (/) placed in a format specification terminates input or output for the current record and initiates a new record. For example

```
    WRITE (6,40) K,L,M,N,O,P
40  FORMAT (3I6.6/I6,2F8.4)
```

is equivalent to

```
    WRITE (6,40) K,L,M
40  FORMAT (3I6.6)
    WRITE (6,50) N,O,P
50  FORMAT (I6,2F8.4)
```

On input from a sequential−access file, the current portion of the remaining record is skipped, a new record is read, and the current position is set to the first character of the record. *n* slashes in succession cause *n* - 1 records to be skipped.

On output to a file connected for sequential access, a new record is created and becomes the last and current record of the file. Also, *n* slashes in succession cause *n* - 1 blank lines to be generated.

Through the use of two or more successive slashes in a format specification, entire records can be skipped for input and records containing no characters can be generated for output. If the file is an internal file, or a file connected for direct access, skipped records are filled with blank characters on output.

n slashes at the beginning or end of a format specification result in *n* skipped or blank records. On input and output from a direct−access file, the record number is increased by one and the file is positioned at the beginning of the record that has that record number. This record becomes the current record.

Interaction Between I/O List and Format

The beginning of formatted data transfer using a format specification initiates format control. Each action of format control depends on information jointly provided by

- the next descriptor contained in the format specification
- the next item in the I/O list, if one exists

If an I/O list specifies at least one list item, at least one repeatable descriptor must exist in the format specification. Note that an empty format specification of the form ( ) can be used only if no list items are specified; in this case, one input record is skipped or one output record containing no characters is written.

Except for a field descriptor preceded by a repeat specification, *r ed*, or a format specification preceded by a repeat specification, *r (flist)*, a format specification is interpreted from left to right (see ''Repeat Counts''). Note that an omitted repeat specification is treated the same as a repeat specification whose value is one.

To each repeatable field descriptor interpreted in a format specification, there corresponds one item specified by the I/O list, except that a list item of type complex is treated as two real items when an **F**, **E**, **D**, or **G** field descriptor is encountered. To each **P**, **X**, **T**, **TL**, **TR**, **S**, **SP**, **SS**, **H**, **BN**, **BZ**, slash (/), colon (:), dollar sign ($), or character edit descriptor, there is no corresponding item specified by the I/O list, and format control communicates information directly to the record.

Whenever format control encounters a repeatable edit descriptor in a format specification, it determines whether there is another item in the I/O list. If there is such an item, it transmits appropriately edited information between the item and the record, and then format control proceeds. If there is no other item, format control terminates.

If format control encounters the right most parenthesis of a complete format specification and no items remain in the list, format control terminates. However, if there are more items in the list, the file is positioned at the beginning of the next record, and format control then reverts to the beginning of the format specification terminated by the last preceding right parenthesis ()). If there is no such preceding right parenthesis ()), format control reverts to the first left parenthesis (() of the format specification. If such a reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor (see ''D Field Descriptor'') the **S**, **SP**, or **SS** edit descriptor sign control, or the **BN** or **BZ** edit descriptor blank control.

List−Directed Formatting

List−directed formatting allows formatted input and output without specifying a format specification. An asterisk (*) is used as a format identifier to invoke a list−directed format.

List−directed formatting can be applied to both internal and external files.

List−Directed Input

The characters in one or more list−directed records form a sequence of values and value separators. Each value is either a constant, or a null value or has one of the following forms:

*r*c*
*r**

where

r	is a nonzero, unsigned integer constant denoting the number of successive appearances of *c* or null values.

c is a constant.

The *r** form is equivalent to *r* successive null values. Neither form can contain embedded blanks, except where permitted within the constant *c*.

Data values can be separated with one of the following value separators:

- A comma optionally preceded and followed by one or more contiguous blanks.
- A slash (/) optionally preceded and followed by one or more contiguous blanks. A slash encountered by a list−directed input statement ends the execution of the input statement after assignment of the previous value, if any; any remaining list items are treated as if null values were supplied. A slash is not used as a separator on output.
- One or more contiguous blanks between two constants or following the last constant. Blanks used in the following manner are not treated as part of any value separator in a list−directed input record:
  - blanks within a character constant
  - embedded blanks surrounding the real or imaginary part of a complex constant
  - leading blanks in the first record read by each execution of a list−directed input statement, unless immediately followed by a slash or comma

The end of a record has the effect of a blank, except when it appears within a character constant. Two or more consecutive blanks are treated as a single blank, unless they occur within a character constant.

There are three differences between the input forms acceptable to format specifiers for a data type and those used for list−directed formatting. A data value must have the same type as the list item to which it corresponds. Blanks are not interpreted as zeros. Embedded blanks are only allowed in constants of character or complex type.

Rules governing input forms of list items for list−directed formatting are

- For data of type real or double precision, the input form is the same as a numeric input field for **F** editing that has no fractional part, unless a decimal point appears within the field.
- For data of type complex, the input form consists of an ordered pair of numeric constants separated by a comma and enclosed in a pair of parentheses. The first numeric constant is the real part of the complex value, while the second constant is the imaginary part. Each of the constants representing the real and imaginary parts may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.
- For data of type logical, the input form must not include either slashes or commas among the optional characters allowed for **L** editing.
- For data of type character, the input form is a character constant: a non empty string of characters enclosed in apostrophes or quotation marks. When apostrophes are used as the character constant delimiter, each apostrophe within the apostrophes is represented by a pair of apostrophes without an intervening blank or end of record.

  When quotation marks are used as the character constant delimiter, each quotation mark within the quotation marks is represented by a pair of quotation marks without an intervening blank or end of record. Character constants can be continued on as many records as needed. Constants are assigned to list items as in character assignment statements.

- A null value is specified by two successive value separators, by the *r** form, or by not having any characters before the first value separator in the first record read by the execution of the list–directed statement. A null value has no effect on the corresponding list item. A single null value may represent an entire complex constant but cannot be used as either the real or imaginary part alone.
- You can specify commas as value separators in the input record when executing a formatted read of non character variables. The commas override the field lengths in the input statement. For example,

  ```
  (i10, f20.10,i4)
  ```

  reads the following record correctly:

  ```
  -345,.05e-3,12
  ```

List–Directed Output

The form of the values produced is the same as that required for input, except as noted below:

- Logical output constants are **T** for the value true and **F** for the value false.
- Integer output constants are produced as for an **I***w* edit descriptor, where *w* depends on whether the list item is **INTEGER*2** or **INTEGER*4** type.
- For complex constants, the end of a record will occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record.
- Produced character constants are not delimited by apostrophes or quotation marks, are not preceded or followed by a value separator, and have each internal apostrophe represented externally by one apostrophe and each internal quotation mark represented by one quotation mark. A blank character for carriage control is inserted at the beginning of a record containing the continuation of a character constant.
- Slashes and null values are not produced, but each record begins with a blank character to provide carriage control if the record is printed.
- Two noncharacter values in succession in the same record will be separated by a value separator of one or more blanks. No value separator is produced before or after a character value.

Chapter 10

Statement Functions and Subprograms

This chapter contains the following subsections:

- "Overview"
- "Statement Functions"
- "Parameter Passing"
- "Function and Subroutine Subprograms"
- "FUNCTION"
- "SUBROUTINE"
- "ENTRY"
- "INCLUDE"

Statement functions and subprograms are program units that receive control when referenced or called by a statement in a main program or another subprogram. A subprogram is either written by the user or supplied with the Fortran compiler. This chapter discusses user−written subprograms; compiler−supplied functions and subroutines are discussed in Appendix A, "Intrinsic Functions."

Overview

This chapter explains the syntax and rules for defining three types of program units:

- *Statement functions* consist of a single arithmetic statement defined within the main program unit or a subprogram.
- *Function subprograms* consist of one or more statements defined external to the main program unit. They are invoked when referenced as a primary in an expression contained in another program unit.
- *Subroutine subprograms* consist of one or more program statements defined as external to the main program unit. It is invoked when referenced in a **CALL** statement (See Chapter 6, "Control Statements") in another program unit.

This chapter also explains the syntax and rules for the **FUNCTION**, **SUBROUTINE**, **ENTRY**, and **INCLUDE** statements, that are used to specify function and subroutine subprograms.

Statement Functions

A statement function definition is similar in form to an arithmetic, logical, or character assignment statement. The name of a statement function is local to the program unit in which it is defined. A statement function definition must appear only after the specification statements and before the first executable statement of the program unit in which it appears.

Defining a Statement Function

A statement function statement has the form

fun `([`*d* `[,`*d*`]...]) =` *e*

where

fun	is a symbolic name of the function.
d	is a dummy argument.
e	is an expression.

Each dummy argument *d* is a variable name called a statement function dummy argument. The statement function dummy argument list indicates the order, number, and type of arguments for the statement function. All arguments need not have the same data type. A specific dummy argument may appear only once in the list. A variable name that serves as a dummy argument can also be the name of a local variable or common block in the same program unit.

Each primary of the expression *e* can include

- constants
- symbolic names of constants
- variable references
- array element references
- library function references
- references to other statement functions
- function subprogram references
- dummy subprogram references
- an expression composed of the above forms and enclosed in parentheses

If a statement function dummy argument name is the same as the name of another entity, the appearance of that name in the expression of a function is a reference to the statement function dummy argument. A dummy argument that appears in a **FUNCTION** or **SUBROUTINE** statement may be referenced in the expression of a function statement with the subprogram.

A dummy argument that appears in an **ENTRY** statement may be referenced in the expression of the statement function only if the dummy argument name appears in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement preceding the statement function definition.

Referencing a Statement Function

A statement function is referenced by using its name with actual arguments, if any, enclosed in parentheses. The form of a statement function reference is

fun`([`*exp*`[,`*exp*`]...])`

where

fun is a statement function name.

exp is an expression.

Operational Conventions and Restrictions

Expressions must agree in order, number, and type with the corresponding dummy arguments. An expression can be any expression except a character expression involving concatenation in which the length attribute of one of the operands is specified with an asterisk.

Execution of a statement function reference results in

- Evaluation of actual arguments (*exp*) that are expressions.
- Association of actual arguments with their corresponding dummy arguments.
- Evaluation of the expression e in the statement function definition.
- Type conversion of the resulting value to the data type of the function, if necessary. This value is returned as the value of the statement function reference.
- A statement function can be referenced only in the program unit that contains its definition. A statement function can reference another statement function that has been defined before the referencing function but not one that is defined after the referencing function.
- A statement function name is local to the program unit and must not be used as the name of any other entity in the program unit except the name of a common block.
- The symbolic name used to identify a statement function cannot appear as a symbolic name in any specification statement except a type statement (to specify the type of the function) or as the name of a common block in the same program unit.
- A dummy argument of a statement function must not be redefined or become undefined through a function subprogram reference in the expression of a statement function.
- The symbolic name of a statement function cannot be an actual argument and must not appear in an **EXTERNAL** statement.
- A statement function in a function subprogram cannot contain a function reference to the name of an entry to the function subprogram.
- The length specification of a statement function dummy argument of type character must be an integer constant.

Parameter Passing

Parameter passing involves function and subroutine arguments. This section explains the difference between actual and dummy arguments. It also describes the Fortran extensions known as built−in functions.

Arguments

Dummy arguments are used in function subprograms, subroutine programs, and statement functions to indicate the types of actual arguments and whether each argument is a single value, an array of values, a subprogram, or a statement label. Dummy argument names must not appear in **EQUIVALENCE**, **DATA** , **PARAMETER**, **SAVE**, **INTRINSIC**, or **COMMON** statements, except as common block names. Dummy argument names must not be the same as the subprogram names in **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statements in the same program unit.

Actual arguments are the items that are specified in the call to the function. Actual arguments are bound to the corresponding dummy arguments when the subprogram call is reached. Actual arguments can change with each call to the subprogram. Of course, the types of the paired actual argument and the dummy argument must match. The types do not have to match if the actual argument is a subroutine name or an alternate return specifier.

When a function or a subroutine reference is executed, an association is established between the corresponding dummy and actual arguments. The first dummy argument becomes associated with the first actual argument, the second dummy argument becomes associated with the second actual argument, and so on.

An array can be passed to a function or subroutine as an actual argument if the corresponding dummy argument is also an array declared in a **DIMENSION** or type statement but not in a **COMMON** statement. The size of the array in the calling program unit must be smaller than or equal to the size of the corresponding dummy array in the subprogram. The array in the function or subroutine can also have adjustable dimensions.

Built−In Functions

Built−in functions provide communication with non−Fortran programs that require arguments passed in a specific format. (See Chapter 3 of the *Fortran 77 Programmer's Guide* for information about communicating with programs written in the C and Pascal languages.)

Use the built−in functions **%VAL**, **%REF**, and **%DESCR** along with arguments within an argument list. The built−in function **%LOC** is intended for global use.

%VAL

The bulit−in **%VAL** function passes the argument as a 32−bit value; the function extends arguments smaller than 32 bits to 32−bit signed values. The function has the following syntax:

`%VAL(`*a*`)`

where *a* is an argument within an argument list.

%REF

The built−in **%REF** function passes an argument by reference. It has the syntax

`%REF(`*a*`)`

where *a* is an argument within an argument list.

%DESCR

The built−in **%DESCR** function has no functionality, but is included for compatibility with VAX Fortran. It has the syntax

`%DESCR(`*a*`)`

where *a* is an argument within an argument list.

%LOC

The built−in **%LOC** function returns a 32−bit run−time address of its argument. It has the syntax

`%LOC(`*a*`)`

where *a* is an argument whose address is to be returned.

Function and Subroutine Subprograms

A function subprogram consists of a **FUNCTION** statement followed by a program body that terminates with an **END** statement. It has the following characteristics:

- defined external to the main program unit
- referenced as a primary in an expression contained in another program unit
- considered part of the calling program

A Fortran program can call a subroutine subprogram written in any language supported by the RISCompiler System. (See Chapter 3 of the *Fortran 77 Programmer's Guide* for information on writing Fortran programs that interact with programs in other languages.)

A subroutine subprogram consists of a **SUBROUTINE** statement, followed by a program body that terminates with an **END** statement (See Chapter 6, "Control Statements") and is defined external to the main program.

Referencing Functions and Subroutines

A function subprogram is referenced as a primary in an expression, while a subroutine subprogram is referenced with a **CALL** statement (See Chapter 6, "Control Statements") contained in another program. Reference to a function subprogram has the form

fun`([`*a*`[,`*a*`]...])`

where *fun* is a symbolic name of the function subprogram and *a* is an actual argument.

If *fun* is of type character, then its length must not have been specified with an asterisk (*) in the calling subprogram.

You can write subroutines that call themselves either directly or through a chain of other subprograms if the automatic storage of variables is in effect. The **-automatic** command line option (described in Chapter 11, "Compiler Options"), by default, causes the automatic storage of variables.

The actual arguments comprise an argument list and must agree in order, number, and type with the corresponding dummy arguments in the referenced function or subroutine. An actual argument in a function reference must be one of the following:

- an expression, except a character expression, involving concatenation of an operand whose length is specified by an asterisk
- an array name
- an intrinsic function name
- an external function or subroutine name
- a dummy function or subroutine name
- a Hollerith constant

An actual argument may be a dummy argument that appears in a dummy argument list within the subprogram containing the reference.

The use of a dummy name allows actual names to be passed through several levels of program units.

If a Hollerith constant is used as an actual argument in a **CALL** statement, the corresponding dummy argument must not be a dummy array and must be of arithmetic or logical data type.

The same rules apply to the actual arguments in a subroutine reference, except that in addition to the forms described above, the actual dummy argument of a subroutine may be an alternate return specifier. An alternate return specifier has the form *s, where *s* is the statement label of an executable statement appearing in the same program unit as the **CALL** statement.

For example,

```
SUBROUTINE MAXX(A,B,*,*,C)
```

The actual argument list passed in the **CALL** must include alternate return arguments in the corresponding positions of the form *s. The value specified for s must be the label of an executable statement in the program unit that issued the call.

An actual argument can also be omitted by specifying only the comma delimiters without an argument in between. In this case, the omitted argument is treated as if it were **%VAL** (0).

Note that the use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type. If an external function or subroutine or dummy name is used as an actual argument, the name must appear in an **EXTERNAL** statement. If an intrinsic name is used as an actual argument, the name must appear in an **INTRINSIC** statement and must be one of those listed in Appendix A, "Intrinsic Functions," as a specific name. It must not be one of the intrinsics for type conversion, for choosing the largest or smallest value, or for lexical relationship.

Executing Functions and Subroutines

Execution of an reference to a function subprogram and subroutine subprogram results in

- evaluation of expressions that constitute actual arguments

- association of actual arguments from the calling program unit with the corresponding dummy arguments in the subprogram
- execution of the statements comprising the subprogram based on the execution control sequence of the program unit
- return of program control to the calling program unit when either a **RETURN** statement is encountered or the execution control flows into the **END** statement

The name of a function subprogram must appear as a variable at least once in the subprogram and must be defined at least once during each subprogram execution. Once the variable is defined, it may be referenced elsewhere in the subprogram and become redefined. When program control is returned to the calling program, this value is returned as the value of the function reference. If this variable is a character variable with a length specified by an asterisk, it may not appear as an operand in a concatenation operation but can be defined in an assignment statement.

A subroutine does not return an explicit value to the point of invocation in the calling program unit. However, both the subroutine and the function can return values to the calling program unit by defining their dummy arguments during execution.

FUNCTION

The **FUNCTION** statement is the first statement of a function subprogram. It specifies the symbolic name of the function and its type.

Syntax

`[`*typ*`] FUNCTION` *fun* `[*`*len*`] ([`*d*`[,`*d*`]...])`

where

typ — optionally specifies the data type of the function name, which determines the value returned to the calling program. The following forms for *typ* are allowed:

INTEGER	DOUBLE PRECISION	LOGICAL
INTEGER*2	COMPLEX	LOGICAL*1
INTEGER*4	COMPLEX*8	LOGICAL*2
REAL	COMPLEX*16	LOGICAL*4
REAL*4	DOUBLE COMPLEX	CHARACTER [**len*]
REAL*8		

fun — is a symbolic name of the function subprogram in which the **FUNCTION** statement appears.

len — specifies the length of the data type; fun must be a nonzero, unsigned constant. Do not specify *len* when the function is type **CHARACTER** with an explicit length following the keyword **CHARACTER**.

w — is a dummy argument and can be a variable, array name, or dummy subprogram name.

Rules for Use

- A **FUNCTION** statement must appear only as the first statement of a function subprogram.
- The type specification may be omitted from the **FUNCTION** statement, and the function name may be specified in a type statement in the same program unit. If neither of these options is used, the function is implicitly typed.
- The symbolic name of a function is a global name and must not be the same as any other global or local name, except a variable name, in the function subprogram.
- If the function type is specified in the **FUNCTION** statement, the function name must not appear in a type statement.
- In the type specification **CHARACTER**, *len* can have any of the forms allowed in a **CHARACTER** statement, except that an integer constant expression must not include the symbolic name of a constant. If the name of the function is type character, then each entry name in the function subprogram must be type character. If the length is declared as an asterisk, all such entries must have a length declared with an asterisk.
- A function specified as a subprogram may be referenced within any other subprogram or in the main program of the executable program.

Restrictions

- A function subprogram cannot contain a **BLOCK DATA**, **SUBROUTINE**, or **PROGRAM** statement.
- A function name cannot have its type explicitly specified more than once in a program unit.
- In a function subprogram, a dummy argument name cannot appear in an **EQUIVALENCE**, **PARAMETER**, **SAVE**, **INTRINSIC**, **DATA**, or **COMMON** statement, except as a common block name.
- A character dummy argument with a length specified as an asterisk must not appear as an operand for concatenation, except in a character assignment statement.
- The compiler system permits recursion if the automatic storage of variables is in effect. The **-automatic** command line option (See Chapter 11, "Compiler Options,"), by default, causes the automatic storage of variables.

SUBROUTINE

A **SUBROUTINE** statement must be the first statement of a subroutine subprogram.

Syntax

`SUBROUTINE` *sub*`[( [`*d* `[,`*d*`]...] )]`

where

sub — is a symbolic name of the subroutine program unit.

d — is a dummy argument and may be a variable name, array name, dummy subprogram name, or asterisk. The asterisk denotes an alternate return.

Rules for Use

- A **SUBROUTINE** statement must be the first statement of a subroutine subprogram.
- If there are no dummy arguments, use either of the following forms:

  ```
  SUBROUTINE sub
  SUBROUTINE sub()
  ```

- One or more dummy arguments can become defined or redefined to return results.
- The symbolic name of a subroutine is global and cannot be the same as any other global or local name in the program unit.
- A **CALL** statement within the body of a subroutine may reference the subroutine itself (recursion) if the automatic storage attribute is specified. See Chapter 4, "Specification Statements," for more information.

Restrictions

- A subroutine subprogram cannot contain a **BLOCK DATA**, **FUNCTION**, or **PROGRAM** statement.
- In a subroutine, a dummy argument name is local to the program unit and cannot appear in an **EQUIVALENCE**, **SAVE**, **INTRINSIC**, **DATA**, or **COMMON** statement, except as a common block name.
- A character dummy argument whose length is specified as an asterisk cannot appear as an operand for concatenation, except in a character assignment statement.

ENTRY

The **ENTRY** statement specifies a secondary entry point in a function or subroutine subprogram. It allows a subprogram reference to begin with a particular executable statement within the function or subroutine subprogram in which the **ENTRY** statement appears.

Syntax

```
ENTRY en[( [d[,d]...] )]
```

where

en — is a symbolic name of the entry point.

d is a dummy argument.

If there are no dummy arguments, you can use the following forms:

```
ENTRY en
ENTRY en()
```

Method of Operation

Each **ENTRY** statement in a function or subroutine provides an additional name you can use to invoke that subprogram. When you invoke it with one of these names, it begins execution at the first executable statement following the entry statement that provided the name.

Within a function, each of its names (the one provided by the **FUNCTION** statement, plus the ones provided by the **ENTRY** statements) acts like a variable. By the time the function returns, you must have defined the function return value by assigning it to one of these variables.

If any of these variables is of type character, all must be of type character; otherwise, the variables need not all have the same data type. Such variables are in effect equivalenced, and therefore

- You need not assign the return value to the name you used to invoke the function; instead, you can assign it to any of the names of the same data type.
- If you assign the return value a name that does not have the same data type as the one you used to invoke the function, then the return value becomes undefined.

Rules for Use

- The **ENTRY** statement may appear anywhere within a function subprogram after the **FUNCTION** statement or within a subroutine after a **SUBROUTINE** statement.
- A subprogram can have one or more **ENTRY** statements.
- The entry name *en* in a function subprogram can appear in a type statement.
- In a function, a local variable with the same name as one of the entries can be referenced.
- A subprogram can call itself directly if the automatic storage of variables is in effect. The **-automatic** command line option (See Chapter 11, "Compiler Options"), by default, causes the automatic storage of variables.
- The order, number, type, and names of the dummy arguments in an **ENTRY** statement can be different from the dummy arguments in the **FUNCTION**, **SUBROUTINE**, or other **ENTRY** statements in the same subprogram. However, each reference to a function or subroutine must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement.

Restrictions

- An **ENTRY** statement must not appear between a block **IF** statement and its corresponding **END IF** statement or between a **DO** statement and the terminal statement of the **DO** loop.

- Within a subprogram, an entry name may not also serve as a dummy argument in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement or be in an **EXTERNAL** statement.
- In a function subprogram, an entry name may also be a variable name provided the variable name is not in any statement (except a type statement) preceding the **ENTRY** statement of that name. After the **ENTRY** statement, the name can be used as a variable name.
- In a function subprogram, if an entry name is of type character, each entry name and the name of the function subprogram must also be of type character and must have the same length declared. If any are of length (*), then *all* must be of length (*).
- In a subprogram, a name that appears as a dummy argument in an **ENTRY** statement is subject to the following restrictions:
  - It must not appear in an executable statement preceding that **ENTRY** statement unless it also appears in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement preceding the executable statement.
  - It must not appear in the expression of a statement function unless the name is also a dummy argument of the statement function. It can appear in a **FUNCTION** or **SUBROUTINE** statement or in an **ENTRY** statement preceding the statement function.

INCLUDE

The **INCLUDE** statement incorporates the contents of a designated file into the Fortran compilation directly following this statement.

Syntax

`INCLUDE "`*filename*`"`

where *filename* is a character string constant that specifies the file to be included.

Rules for Use

- An **INCLUDE** statement can appear anywhere within a program unit.
- On encountering an **INCLUDE** statement, the compiler stops reading statements from the current file and reads the statements in the included file. At the end of the included file, the compiler resumes reading the current file with the statement following the **INCLUDE** statement.

Search Path

On encountering an **INCLUDE** statement, the compiler first searches for a file called *filename*, then for a file named **/usr/include/***filename.*

Restrictions

- An included file or module cannot begin with a continuation line. Each Fortran statement must be completely contained within a single file.

- An **INCLUDE** statement cannon contain continuation lines. The first non comment line following the **INCLUDE** statement cannot be a continuation line.
- An **INCLUDE** statement cannot be labeled. It must not have a statement number in the statement number field.

Chapter 11

Compiler Options

This chapter contains the following subsections:

- "OPTIONS Statement"
- "In−Line Options"
- "$INCLUDE Statement"

This chapter describes options that affect source programs both during compilation and at run time. Execute these options using

- **OPTIONS** statement—specified in the source code as the first statement of a program unit
- In−line options—individual statements embedded in the source code
- **$INCLUDE** statement—includes Fortran source statements from an external library into a program

This chapter discusses these options according to the mechanisms used to specify them. The command line options, which are parameters specified as part of the *f77* command when the compiler is invoked, are explained in the *Fortran 77 Programmer's Guide.*

OPTIONS Statement

The **OPTIONS** statement has the following syntax:

`OPTIONS` *option*`[` *option*`...]`

where *option* can be any of the following:

`/I4    /NOI4    /F77    /NOF77    /CHECK=BOUNDS    /CHECK=NOBOUNDS`
`/EXTEND_SOURCE    /NOEXTEND_SOURCE`

These options perform the same function as the like−named command line options. See Chapter 1 of the *Fortran 77 Programmer's Guide* for a description of these options. Specifying *option* overrides a command line option when they are the same. *option* must always be preceded by a slash (/).

Use the following rules when specifying an **OPTIONS** statement:

- The statement must be the first statement in a program unit and must precede the **PROGRAM**, **SUBROUTINE**, **FUNCTION**, and **BLOCKDATA** statements.
- *option* remains in effect only for the duration of the program unit in which it is defined.

In−Line Options

The syntax for in−line compiler options consists of a dollar sign ($) in column 1 of a source record, followed by the name of the compiler option in either uppercase or lowercase, with no intervening blanks or other separators.

When an in−line compiler option is encountered in a source file, that option is put into effect beginning with the source statement following the in−line compiler option. The sections that follow describe the in−line compiler options supported by the compiler.

The compiler does not support the following options, but, for compatibility with other compilers, it does recognize them:

ARGCHECK	NOTBINARY
BINARY	SEGMENT
CHAREQU	SYSTEM
NOARGCHECK	XREF

When it encounters one of these options, the compiler issues a warning message and treats it as a comment line.

$COL72 Option

The **$COL72** option instructs the compiler to process all subsequent Fortran source statements according to the fixed−format 72−column mode described under Source Program Lines in Chapter 1. The compiler command line option **-col72** has an identical effect on a global basis.

$COL120 Option

The **$COL120** option instructs the compiler to process all subsequent Fortran source statements according to the fixed−format 120−column mode. The compiler command line option **-col120** has an identical effect on a global basis.

$F66DO Option

The **$F66DO** option instructs the compiler to process all subsequent **DO** loops according to the rules of Fortran 66. This principally means that all **DO** loop bodies will be performed at least once regardless of the loop index parameters. The compiler command line option **-onetrip** has the identical effect on a global basis.

$INT2 Option

The **$INT2** option instructs the compiler to make **INTEGER*2** the default integer type and **LOGICAL*1** the default logical type. This convention stays in effect for the remainder of the program and involves any symbolic names that are assigned a data type either by implicit typing rules or by using **INTEGER** or **LOGICAL** declaration statements without a type length being specified. This option is similar to the **-i2** command line option except for the effect on the default logical type.

$LOG2 Option

The **$LOG2** option instructs the compiler to make **LOGICAL*2** instead of **LOGICAL*4** the default type for **LOGICAL**. This convention stays in effect for the remainder of the program and involves any symbolic names that are assigned a data type either by implicit typing rules or by using the **LOGICAL** declaration statement without a type length being specified.

$INCLUDE Statement

The **$INCLUDE** statement includes source lines from secondary files in the current primary source program. This feature is especially useful when two or more separately compiled source programs require an identical sequence of source statements (for example, data declaration statements).

The form of the **$INCLUDE** statement is

`$INCLUDE` *filename*

where *filename* is either an absolute or relative UNIX file name. If the filename is relative and no file exists by that name relative to the current working directory, an error is given and no attempt is made to search an alternative path. The material introduced into the source program by the **$INCLUDE** statement will follow the **$INCLUDE** statement, beginning on the next line. Nesting of **$INCLUDE** statements is permitted within the constraints of the operating system.

Appendix A

Intrinsic Functions

This appendix contains the following subsections:

- "Generic and Specific Names"
- "Operational Conventions and Restrictions"
- "List of Functions"

This appendix describes the intrinsic functions provided with Fortran. Fortran intrinsic functions are identified by two categories of names: specific and generic. An **IMPLICIT** statement does not change the data type of an intrinsic function.

Generic and Specific Names

A *generic name* is the name given to a class of objects. Intrinsic functions that perform the same mathematical function, such as square root, are given a single name. For example, the generic name of the square root function is **SQRT**; this function has four specific names for different data types: **SQRT**, **DSQRT**, **CSQRT**, and **ZSRT** (see Table A−1). However, you can use the generic name **SQRT** regardless of the data type of the arguments.

An intrinsic function preceded by the letters **CD** is equivalent to the generic function with the same base name, except that the arguments must be of type **DOUBLE COMPLEX**.

Intrinsic functions starting with **II** are equivalent to generic functions with the same base name, except that the arguments must of type **INTEGER*2**. Similarly, arguments to intrinsic functions starting with **JI** must be type **INTEGER*4**: for example, **IIAND**, **IIQINT**, **IIQNNT**, **JIQINT**, **JIQNNT**.

When a generic name is referenced, the processor substitutes a function call for a specific name, depending on the data type of the arguments. In this way, the same name can be used for different types of arguments.

When an intrinsic function is to be used as the actual argument to another function, you must always use the specific name, never the generic name.

If a generic name is referenced, the type of the result is the same as the type of the argument, except for functions performing type conversion, nearest integer, and absolute value with a complex argument. Some intrinsic functions allow more than one argument, in which case all the arguments must be of the same type so that the function can decide which specific name function it should use.

If the specific name or generic name appears as the dummy argument of a function or subroutine, that symbolic name cannot identify an intrinsic function in that program unit.

A name in an **INTRINSIC** statement must be the specific name or generic name of an intrinsic function, as given in Table A−1

Referencing an Intrinsic Function

Reference an intrinsic function in the form

fun (*a*[,*a*]...)

where

fun is the generic or specific name of the intrinsic function.

a is an actual argument.

The actual arguments (*a*) constitute the argument list and must agree in order, number, and type with the specification described in this appendix and with each other. Each argument can be any expression. The expression cannot contain an concatenation in which one or more of the operand lengths are specified with an asterisk.

A function reference can be used as a primary in an expression. The following example involves referencing an intrinsic function:

```
X = SQRT(B**2-4*A*C)
```

The result of a function becomes undefined when its arguments are not mathematically defined or exceed the numeric range of the processor.

Operational Conventions and Restrictions

For most intrinsic functions, the data type of the result of the intrinsic function is the same as the arguments. If two or more arguments are required or permitted, then all arguments must be of the same type. An **IMPLICIT** statement does not change the data type of a specific or generic name of an intrinsic function.

If an intrinsic function name is used as an actual argument in an external procedure reference, the name must be one of the specific names and must appear in an **INTRINSIC** statement. However, names of intrinsic functions for type conversion, for lexical relationship, and for choosing the smallest or largest value cannot be used as actual arguments.

List of Functions

Table A−1lists the available intrinsic functions. Operational conventions and restrictions (other than those already given) are listed at the end of the table.

Note: **REAL*16** intrinsic functions are not supported. The compiler issues a warning message when the name of a **REAL*16** intrinsic function is encountered; the equivalent double precision (**REAL*8**) function is used instead.

Table A−1 Intrinsic Functions

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to INTEGER	1	INT[a]		INTEGER*1	INTEGER*2
				INTEGER*1	INTEGER*4
				INTEGER*1	INTEGER*8
				INTEGER*2	INTEGER*4

				INTEGER*2	INTEGER*8
				INTEGER*4	INTEGER*4
				INTEGER*4	INTEGER*8
				INTEGER*8	INTEGER*8
			IINT	REAL*4	INTEGER*2
			JINT	REAL*4	INTEGER*4
			KINT	REAL*4	INTEGER*8
			IIDINT	REAL*8	INTEGER*2
			JIDINT	REAL*8	INTEGER*4
			KIDINT	REAL*8	INTEGER*8
				COMPLEX*8	INTEGER*2
				COMPLEX*8	INTEGER*4
				COMPLEX*8	INTEGER*8
				COMPLEX*16	INTEGER*2
				COMPLEX*16	INTEGER*4
				COMPLEX*16	INTEGER*8
		SHORT		INTEGER*1	INTEGER*2
				INTEGER*2	INTEGER*2
				INTEGER*4	INTEGER*2
				REAL*4	INTEGER*2
				REAL*8	INTEGER*2
				COMPLEX*8	INTEGER*2
				COMPLEX*16	INTEGER*2
		LONG		INTEGER*1	INTEGER*4
				INTEGER*2	INTEGER*4
				INTEGER*4	INTEGER*4
				REAL*4	INTEGER*4
				REAL*8	INTEGER*4
				COMPLEX*8	INTEGER*4
				COMPLEX*16	INTEGER*4
	1	IFIX	IIFIX	REAL*4	INTEGER*2
			JIFIX	REAL*4	INTEGER*4
			KIFIX	REAL*4	INTEGER*8
		IDINT	IIDINT	REAL*8	INTEGER*2
			JIDINT	REAL*8	INTEGER*4
			KIDINT	REAL*8	INTEGER*8
Truncation		AINT	AINT	REAL*4	REAL*4
			DINT	REAL*8	REAL*8
Conversion to	1	REAL		INTEGER*1	REAL*4
REAL			FLOATI	INTEGER*2	REAL*4
			FLOATJ	INTEGER*4	REAL*4
			FLOATK	INTEGER*8	REAL*4
				REAL*4	REAL*4
			SNGL	REAL*8	REAL*4
				COMPLEX*8	REAL*4
				COMPLEX*16	REAL*4
	1	FLOAT		INTEGER*1	REAL*4
			FLOATI	INTEGER*2	REAL*4
			FLOATJ	INTEGER*4	REAL*4
			FLOATK	INTEGER*8	REAL*4
	1	SNGL		INTEGER*1	REAL*4
			FLOATI	INTEGER*2	REAL*4

			FLOATJ	INTEGER*4	REAL*4
			FLOATK	INTEGER*8	REAL*4
			REAL	REAL*4	REAL*4
				REAL*8	REAL*4
Conversion to DOUBLE PRECISION	1	DBLE		INTEGER*1	REAL*8
				INTEGER*2	REAL*8
				INTEGER*4	REAL*8
				INTEGER*8	REAL*8
			DBLE	REAL*4	REAL*8
				REAL*8	REAL*8
				COMPLEX*8	REAL*8
				COMPLEX*16	REAK*8
		DFLOAT		INTEGER*1	REAL*8
			DFLOTI	INTEGER*2	REAL*8
			DFLOTJ	INTEGER*4	REAL*8
			DFLOTK	INTEGER*8	REAL*8
			DFLOATK	INTEGER*8	REAL*8
Conversion to COMPLEX	1, 2	CMPLX		INTEGER*1	COMPLEX*8
	1,2			INTEGER*2	COMPLEX*8
	1,2			INTEGER*4	COMPLEX*8
	1,2			INTEGER*8	COMPLEX*8
	1,2			REAL*4	COMPLEX*8
	1,2			REAL*8	COMPLEX*8
	1			COMPLEX*8	COMPLEX*8
	1			COMPLEX*16	COMPLEX*8
Complex Conjugate	1	CONJG	CONJG	COMPLEX*8	COMPLEX*8
			DCONJG	COMPLEX*8	COMPLEX*16
Conversion to DOUBLE COMPLEX	1, 2	DCMPLX		INTEGER*1	COMPLEX*16
	1,2			INTEGER*2	COMPLEX*16
	1,2			INTEGER*4	COMPLEX*16
	1,2			INTEGER*8	COMPLEX*16
	1,2			REAL*4	COMPLEX*16
	1,2			REAL*8	COMPLEX*16
	1			COMPLEX*8	COMPLEX*16
	1			COMPLEX*16	COMPLEX*16
Conversion to character	1		CHAR	LOGICAL*1	CHARACTER
				INTEGER*1	CHARACTER
				INTEGER*2	CHARACTER
				INTEGER*4	CHARACTER
				INTEGER*8	CHARACTER
Maximum value	2 or more	MAX		INTEGER*1	INTEGER*1
			IMAX0	INTEGER*2	INTEGER*2
			JMAX0	INTEGER*4	INTEGER*4
			KMAX0	INTEGER*8	INTEGER*8
			AMAX1	REAL*4	REAL*4
			DMAX1	REAL*8	REAL*8
		MAX0		INTEGER*1	INTEGER*1
			IMAX0	INTEGER*2	INTEGER*2
			JMAX0	INTEGER*4	INTEGER*4
			KMAX0	INTEGER*8	INTEGER*8
		MAX1	IMAX1	REAL*4	INTEGER*2
			JMAX1	REAL*4	INTEGER*4

			KMAX1	REAL*4	INTEGER*8
		AMAX0		INTEGER*1	REAL*4
			AIMAX0	INTEGER*2	REAL*4
			AJMAX0	INTEGER*4	REAL*4
			AKMAX0	INTEGER*8	REAL*4
Minimum value	2 or more	MIN		INTEGER*1	INTEGER*1
			IMIN0	INTEGER*2	INTEGER*2
			JMIN0	INTEGER*4	INTEGER*4
			KMIN0	INTEGER*8	INTEGER*8
			AMIN1	REAL*4	REAL*4
			DMIN1	REAL*8	REAL*8
		MIN0		INTEGER*1	INTEGER*1
			IMIN0	INTEGER*2	INTEGER*2
			JMIN0	INTEGER*4	INTEGER*4
			KMIN0	INTEGER*8	INTEGER*8
		MIN1	IMIN1	REAL*4	INTEGER*2
			JMIN1	REAL*4	INTEGER*4
			KMIN1	REAL*4	INTEGER*8
		AMIN0		INTEGER*1	REAL*4
			AIMIN0	INTEGER*2	REAL*4
			AJMIN0	INTEGER*4	REAL*4
			AKMIN0	INTEGER*8	REAL*4
Nearest integer	1	NINT[b]	ININT	REAL*4	INTEGER*2
			JNINT	REAL*4	INTEGER*4
			KNINT	REAL*4	INTEGER*8
			IIDNNT	REAL*8	INTEGER*2
			JIDNNT	REAL*8	INTEGER*4
			KIDNNT	REAL*8	INTEGER*8
		ANINT	ANINT	REAL*4	REAL*4
			DNINT	REAL*8	REAL*8
		IDNINT	IIDNNT	REAL*8	INTEGER*2
			JIDNNT	REAL*8	INTEGER*4
			KIDNNT	REAL*8	INTEGER*8
Zero−Extend functions	1	ZEXT	IZEXT	LOGICAL*1	INTEGER*2
				LOGICAL*2	INTEGER*2
				INTEGER*1	INTEGER*2
				INTEGER*2	INTEGER*2
			JZEXT	LOGICAL*1	INTEGER*4
				LOGICAL*2	INTEGER*4
				LOGICAL*4	INTEGER*4
				INTEGER*1	INTEGER*4
				INTEGER*2	INTEGER*4
				INTEGER*4	INTEGER*4
			KZEXT	LOGICAL*1	INTEGER*8
				LOGICAL*2	INTEGER*8
				LOGICAL*4	INTEGER*8
				LOGICAL*8	INTEGER*8
				INTEGER*1	INTEGER*8
				INTEGER*2	INTEGER*8
				INTEGER*4	INTEGER*8
				INTEGER*8	INTEGER*8

Absolute value	1	ABS		INTEGER*1	INTEGER*1
			IIABS	INTEGER*2	INTEGER*2
			JIABS	INTEGER*4	INTEGER*4
			KIABS	INTEGER*8	INTEGER*8
			ABS	REAL*4	REAL*4
			DABS	REAL*8	REAL*8
			CABS	COMPLEX*8	COMPLEX*8
			CDABS	COMPLEX*16	COMPLEX*16
			ZABS	COMPLEX*16	COMPLEX*16
		IABS[c]		INTEGER*1	INTEGER*1
			IIABS	INTEGER*2	INTEGER*2
			JIABS	INTEGER*4	INTEGER*4
			KIABS	INTEGER*8	INTEGER*8
Remaindering	2	MOD[d]		INTEGER*1	INTEGER*1
			IMOD	INTEGER*2	INTEGER*2
			JMOD	INTEGER*4	INTEGER*4
			KMOD	INTEGER*8	INTEGER*8
			AMOD	REAL*4	REAL*4
			DMOD	REAL*8	REAL*8
Transfer of sign	2	SIGN		INTEGER*1	INTEGER*1
			IISIGN	INTEGER*2	INTEGER*2
			JISIGN	INTEGER*4	INTEGER*4
			KISIGN	INTEGER*8	INTEGER*8
			SIGN	REAL*4	REAL*4
			DSIGN	REAL*8	REAL*8
		ISIGN[e]		INTEGER*1	INTEGER*1
			IISIGN	INTEGER*2	INTEGER*2
			JISIGN	INTEGER*4	INTEGER*4
			KISIGN	INTEGER*8	INTEGER*8
Positive difference	2	DIM		INTEGER*1	INTEGER*1
			IIDIM	INTEGER*2	INTEGER*2
			JIDIM	INTEGER*4	INTEGER*4
			KIDIM	INTEGER*8	INTEGER*8
			DIM	REAL*4	REAL*4
			DDIM	REAL*8	REAL*8
		IDIM		INTEGER*1	INTEGER*1
			IIDIM	INTEGER*2	INTEGER*2
			JIDIM	INTEGER*4	INTEGER*4
			KIDIM	INTEGER*8	INTEGER*8
DOUBLE PRECISION product of REALs	2		DPROD	REAL*4	REAL*4
Length of character entry	1		LEN	CHARACTER	INTEGER*4
Index of a substring	2		INDEX[f]	CHARACTER	INTEGER*4
Character (ASCII value of 1−byte character argument)	1		ICHAR	CHARACTER	INTEGER*2
				CHARACTER	INTEGER*4
				CHARACTER	INTEGER*8
Logically greater than or equal	2		LGE	CHARACTER	LOGICAL*4

Logically greater than	2		LGT	CHARACTER	LOGICAL*4
Logically less than or equal	2		LLE	CHARACTER	LOGICAL*4
Logically less than	2		LLT[g]	CHARACTER	LOGICAL*4
Imaginary part of complex number	1	IMAG	AIMAG DIMAG	COMPLEX*8 COMPLEX*16	REAL*4 REAL*8
Real part of complex number	1		REAL DREAL	COMPLEX*8 COMPLEX*16	REAL*4 REAL*8
Square root	1	SQRT	SQRT[h] DSQRT CSQRT CDSQRT ZSQRT	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16
Exponential	1	EXP	EXP DEXP CEXP CDEXP ZEXP	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16
Natural logarithm	1	LOG	ALOG[i] DLOG CLOG CDLOG ZLOG	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16
Common logarithm	1	LOG10	ALOG10 DLOG10	REAL*4 REAL*8	REAL*4 REAL*8
Sine	1	SIN	SIN DSIN CSIN CDSIN ZSIN	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16
Sine (degree)	1	SIND[j]	SIND DSIND	REAL*4 REAL*8	REAL*4 REAL*8
Cosine	1	COS	COS DCOS CCOS CDCOS ZCOS	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16 COMPLEX*16
Cosine (degree)	1	COSD	COSD DCOSD	REAL*4 REAL*8	REAL*4 REAL*8
Tangent	1	TAN	TAN DTAN	REAL*4 REAL*8	REAL*4 REAL*8
Tangent (degree)	1	TAND	TAND DTAND	REAL*4 REAL*8	REAL*4 REAL*8
Arcsine	1	ASIN[k,l,m]	ASIN DASIN	REAL*4 REAL*8	REAL*4 REAL*8
Arcsine (degree)	1	ASIND[n]	ASIND DASIND	REAL*4 REAL*8	REAL*4 REAL*8

Arccosine	1	ACOS	ACOS	REAL*4	REAL*4
			DACOS	REAL*8	REAL*8
Arccsine (degree)	1	ACOSD	ACOSD	REAL*4	REAL*4
			DACOSD	REAL*8	REAL*8
Arctangent	1	ATAN[o]	ATAN	REAL*4	REAL*4
			DATAN	REAL*8	REAL*8
Arctangent (degree)	1	ATAND[p]	ATAND	REAL*4	REAL*4
			DATAND	REAL*8	REAL*8
Arctangent	2	ATAN2[q,r]	ATAN2	REAL*4	REAL*4
			DATAN2	REAL*8	REAL*8
Arctangent (degree)	2	ATAN2D	ATAN2D	REAL*4	REAL*4
			DATAN2D	REAL*8	REAL*8
Hyperbolic sine	1	SINH	SINH	REAL*4	REAL*4
			DSINH	REAL*8	REAL*8
Hyperbolic cosine	1	COSH	COSH	REAL*4	REAL*4
			DCOSH	REAL*8	REAL*8
Hyperbolic tangent	1	TANH	TANH	REAL*4	REAL*4
			DTANH	REAL*8	REAL*8
Bitwise AND	2	IAND[1]		INTEGER*1	INTEGER*1
			IIAND	INTEGER*2	INTEGER*2
			JIAND	INTEGER*4	INTEGER*4
			KIAND	INTEGER*8	INTEGER*8
Bitwise inclusive OR	2	IOR[1]		INTEGER*1	INTEGER*1
			IIOR	INTEGER*2	INTEGER*2
			JIOR	INTEGER*4	INTEGER*4
			KIOR	INTEGER*8	INTEGER*8
Bitwise complement	1	NOT[1]	INOT	INTEGER*1	INTEGER*1
			JNOT	INTEGER*2	INTEGER*2
			KNOT	INTEGER*4	INTEGER*4
				INTEGER*8	INTEGER*8
Bitwise exclusive OR	2	IEOR[1]		INTEGER*1	INTEGER*1
			IIEOR	INTEGER*2	INTEGER*2
			JIEOR	INTEGER*4	INTEGER*4
			KIEOR	INTEGER*8	INTEGER*8
Bitwise logical shift	2	ISHFT		INTEGER*1	INTEGER*1
			IISHFT	INTEGER*2	INTEGER*2
			JISHFT	INTEGER*4	INTEGER*4
			KISHFT	INTEGER*8	INTEGER*8
Bitwise circular shift	2	ISHFTC		INTEGER*1	INTEGER*1
			IISHFTC	INTEGER*2	INTEGER*2
			JISHFTC	INTEGER*4	INTEGER*4
			KISHFTC	INTEGER*8	INTEGER*8
Bit extraction	3	IBITS		INTEGER*1	INTEGER*1
			IIBITS	INTEGER*2	INTEGER*2
			JIBITS	INTEGER*4	INTEGER*4
			KIBITS	INTEGER*8	INTEGER*8
Bit set	2	IBITSET		INTEGER*1	INTEGER*1
			IIBSET	INTEGER*2	INTEGER*2
			JIBSET	INTEGER*4	INTEGER*4
			KIBSET	INTEGER*8	INTEGER*8

Bit test	2	BTEST		INTEGER*1	LOGICAL*4
			BITEST	INTEGER*2	LOGICAL*2
			BJTEST	INTEGER*4	LOGICAL*4
			BKTEST	INTEGER*8	LOGICAL*8
Bit clear	2	IBCLR		INTEGER*1	INTEGER*1
			IIBCLR	INTEGER*2	INTEGER*2
			JIBCLR	INTEGER*4	INTEGER*4
			KIBCLR	INTEGER*8	INTEGER*8

[a] INT and IFIX return type INTEGER*2 if the **–i2** compile option is in effect; otherwise, the result type is INTEGER*4.

[b] When NINT or IDNINT is specified as an *argument* in a subroutine call or function reference, the compiler supplies either an INTEGER*2 or an INTEGER*4 function depending on the **–i2** command line option (see Chapter 1of the *Fortran 77 Programmer's Guide*).

[c] The IABS, ISIGN, IDIM, and integer MOD intrinsics accept either INTEGER*2 arguments or INTEGER*4 arguments, and the result is the same type.

[d] The result for MOD, AMOD, and DMOD is undefined when the value of the second argument is zero.

[e] If the value of the first argument of ISIGN, SIGN, or DSIGN is zero, the result is zero.

[f] The result of INDEX is an integer value indicating the position in the first argument of the first substring which is identical to the second argument. The result of INDEX('ABCDEF', 'CD'), for example, would be 3. If no substring of the first argument matches the second argument, the result is zero. INDEX and ICHAR return the result type INTEGER*2 if the **–i2** compile option is in effect; otherwise, the result type is INTEGER*4.

[g] The character relational intrinsics (LLT, LGT, LEE, and LGE) return result type LOGICAL*2 if the **$log2** (see Chapter 11) compile option is in effect; otherwise, the result type is LOGICAL*4.

[h] The value of the argument of SQRT and DSQRT must be greater than or equal to zero. The result of CSQRT is the principal value with the real part greater than or equal to zero. When the real part is zero, the imaginary part is greater than or equal to zero.

[i] The argument of ALOG and DLOG must be greater than zero. The argument of CLOG must not be (0.,0.). The range of the imaginary part of the result of CLOG is: $-p <$ imaginary part $<p$.

[j] The argument for SIND, COSD, or TAND must be in degrees and is treated as modulo 360.

[k] The absolute value of the arguments of ASIN, DASIN, ASIND, DASIND, ACOS, DACOS, ACOSD, and DACSOD must be less than or equal to 1.

[l] The range of the result for ASIN and DASIN is $-\pi/2 <$ result $< \pi/2$; the range of the result for DASIN is $0 <$ result $< \pi$; and the range of the result of ACOS and DACOS is less than or equal to one.

[m] The result of ASIN, DASIN, ACOS, and DACOS is in radians.

[n] The result of ASIND, DASIND, ACOS, DACOSD is in degrees.

[o] The result of ATAN, DATAN, ATAN2, and DTAN2 is in radians.

[p] The result of ATAND, DATAND, ATAN2D, and DATAN2D is in degrees.

[q] If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. When the value of the first argument is zero, the result is zero if the second argumemt is positive and P if the second
argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the aboslute value of the result is P/2. Both arguments must not have the value zero.

[r] Note 3 on this page also applies to ATAN2 and DTAN2D, except for the range of the result, which is:
−180 degrees << result << 180 degrees.