FORTRAN 77 Language Reference Manual

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-0710-030

FORTRAN 77 Language Reference Manual

Document Version 3.0

Document Number 007-0710-030

5/90

Technical Publications:

C J Silverio
Claudia Lohnes

Engineering:

Calvin Vu

FORTRAN 77 Language Reference Manual
Document Version 3.0
Document Number 007-0710-030

Silicon Graphics, Inc.
Mountain View, California

Contents

List of Figures

List of Tables

Preface

About This Manual

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 (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.

FORTRAN 77 is referred to as FORTRAN throughout this manual, except where distinctions between FORTRAN 77 and FORTRAN 66 are being discussed specifically.

The compiler has the ability to convert source programs written in VMS FORTRAN into machine programs executable under IRIX.

Standard FORTRAN 77 rules and syntax appear as normal text throughout the manual, whereas text that describes extensions is shaded, as is this paragraph.

Intended Audience

This manual is intended as a reference manual rather than a tutorial, and assumes familiarity with some 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 the following topics:

- 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 systems functions and subroutines that are callable by FORTRAN programs.
- Multiprocessing FORTRAN enhancements.

Refer to the *Languages Programmer's Guide* for information on the following topics:

- 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 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.*

For information on porting FORTRAN code, refer to Chapter 3 of the *Compatibility Guide*.

Organization of Information

The following topics are covered in this manual:

- FORTRAN language elements
- Data types, constants, variables and arrays
- Expressions
- FORTRAN statements, grouped according to the general class of functions they perform:
  - Specification statements
  - Assignment statements
  - Control statements
  - Input/output statements
  - Format specifications

Appendix A contains tables showing the intrinsic functions supported.

Syntax Conventions

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

UPPER CASE	Upper case letters and words are to be written as shown, except where noted otherwise.
lower case	Lower case abbreviations and words represent characters or numerical values that you define. You replace the abbreviation with the defined value.
[]	Brackets are used to indicate optional items.
{}	Braces surrounding two or more items indicate that at least one of the items must be specified.
|	The *or* symbol separates two or more optional items.
...	An ellipsis indicates that the preceding optional items may appear more than once in succession.
()	A pair of parentheses encloses entities and must be written as shown.
(blank)	Blanks have no significance unless otherwise noted.

Below 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 entity *a(d)* is required, and that one or more of *a*(*d*) may be optionally specified. Note that the pair of parentheses ( ) enclosing *d* are required.

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

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

1. FORTRAN Elements and Concepts

1.1 Overview

This chapter provides definitions for the various elements that comprise 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, rules for writing FORTRAN statements, the main structural elements of a program, and the proper order of statements in a program are discussed in this chapter.

1.2 FORTRAN Character Set

The FORTRAN character set consists of 26 upper-case and 26 lower-case letters (*alphabetic* characters), characters 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 as follows:

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
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol
'	Apostrophe
:	Colon
!	Exclamation Point
_	Underscore
"	Quotation Mark

Lower-case alphabetic characters, and the exclamation mark (!), underscore (_), and the double quote (") are extensions to FORTRAN 77.

Digits are interpreted in base 10 when a numeric value is represented.

A special character may serve as an operator, a part of a character constant, a part of a numeric constant, or some other function.

Blank characters may be used freely to improve the appearance and readability of FORTRAN statements. They have no significance in FORTRAN statements, except in the following cases:

- When used in character constants
- When used in H- and character-editing in format specifications
- When used 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 more detail in the appropriate sections.

1.2.1 Escape Sequences

The compiler accepts the following escape sequences for compatibility with C:

Sequence	Meaning
\n	New line
\t	Tab
\b	Backspace
\f	Form feed
\0	Null
\'	Apostrophe (doesn't terminate a string)
\"	Quotation mark (doesn't terminate a string)
\\\\	\\
\x	x represents any other character

Table 1-1. Escape Sequences

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

The compiler aligns character string constants and unequivalenced scalar local character variables on an INTEGER word boundary. It places a null character after each character string constant appearing outside a DATA statement for compatibility with routines written in C.

1.3 Collating Sequence

The collating sequence for letters and digits defines the relationship between them and is used to compare character strings.

The collating sequence is determined as follows:

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

1.4 Symbolic Names

A symbolic name is a sequence of characters used to identify the following user-defined local and global entities:

Local	variable
	constant
	array
	statement function
	intrinsic function
	dummy procedure

Global	common block
	external function
	subroutine
	main program
	block data subprogram

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

1. FORTRAN symbolic names may contain any number of characters, but only the first 32 of these are significant in distinguishing one symbolic name from another. Symbolic names that are used externally (program names, subroutine names, function names, common block names) are limited to 32 significant characters.

2. The inclusion of the special period (.), underscore (_), and dollar ($) characters in symbolic names is an enhancement to FORTRAN 77. In FORTRAN 77, no special characters are allowed.

Examples of valid symbolic names are:

```
CASH  C3P0 R2D2  LONG_NAME  _THIS_
```

Examples of invalid symbolic names in are:

```
X*4     (Contains a special character)
3CASH   (First character is a digit)
```

1.4.1 Scope of Symbolic Names

The rules for determining the scope of symbolic names are as follows:

1. 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. It must not identify another global entity in the same executable program.

2. 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. It must not identify any other local entity in the same program unit.

3. A symbolic name assigned to a global entity in a program unit must not be used for a local entity in the same unit. However, it may be used for a common block name, or an external function name, that appears in a FUNCTION or ENTRY statement.

1.5 Source Program Lines

A source program line can be thought of as a sequence of character positions, called *columns*, numbered consecutively starting from Column 1 on the left. Lines can be classified as comment lines, initial lines, continuation lines, and debugging lines (an extension to FORTRAN 77).

1.5.1 Comments

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

- An upper-case C (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 mark ! at any position of the line.

1.5.2 Debugging Lines

A upper-case D in Column 1 can be specified for debugging purposes; it permits the conditional compilation of source lines in conjunction with the **d_lines** option described in **Chapter 1** of the *FORTRAN 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.

1.5.3 Initial Lines

Initial lines contain the FORTRAN language statements that make up the source program; these statements are described in detail under Program Organization later in this chapter. Each FORTRAN line is divided into the following fields:

- 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 following sections.

1.5.4 Fixed Format

Consider a FORTRAN line to be divided into columns, with one character per column as indicated below:

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 *FORTRAN Programmer's Guide* for details. Several of these options can be specified in-line as described in Chapter 11.

1.5.5 TAB Character Formatting

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

- Type the Statement Label, and follow it with a TAB. If there is no Statement Label, *start* the line with a TAB.

- 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 non-zero digit follows the TAB, the line is interpreted as an initial line.

- In a continuation line beginning with a TAB followed by a non-zero digit, any characters following the digit to the end of the line are a continuation of the current statement.

- TAB-formatted lines don't have preassigned comment fields. All characters to the end of the line are considered part of the statement. However, an exclamation mark (!) can explicitly define a comment field, from the ! to the end of the line.

The rules for TAB formatting can be summarized:

<<*statement label*>TAB<<*statement*> (initial line)

TAB<<*continuation field*><<*statement*> (continuation line)

TAB<<*statement*> (initial line)

Note that while 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.

1.5.6 Continuation Lines

A continuation line is a continuation of 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, an ampersand (&) in Column 1 can also identify a continuation line. When an & is used in Column 1, Columns 2 through the end of the line are considered part of the statement. In FORTRAN 77, any remaining columns (73-...) of a continuation line are not interpreted.

Up to 19 continuation lines in a row are allowed.

1.5.7 Blank Lines

A line that is entirely blank is a comment line. It can improve the readability of a program.

1.6 Program Organization

Program units are made up of FORTRAN statements. A FORTRAN program consists of one or more program units.

1.6.1 FORTRAN Statements

FORTRAN statements are used to form program units. Each statement is written from Column 7 onwards of an initial line and Column 7 onwards of as many as 99 continuation lines.

A statement must not 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. No other statement may have an initial line that contains END as its first three nonblank characters.

All FORTRAN statements, except for assignment and statement function statements, begin with a keyword. A *keyword* is a sequence of characters that identifies the type of FORTRAN statement.

A statement label provides a means of referring 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"
```

Two or more statements in a program unit must not have the same statement label.

It is not necessary to label a FORTRAN statement. However, only labeled statements may be referenced by other FORTRAN statements. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA, and INCLUDE statements must not be labeled.

1.6.2 Program Units

A program unit consists of a sequence of statements and optional comment lines. It may be a main program or a subprogram. The program unit defines the scope for symbolic names and statement labels.

A program unit always has an END statement as its last statement.

1.6.3 Main Program

The main program is the program unit that initially receives control upon execution.

A main program may have a PROGRAM statement as its first statement. It may 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 may be a non-FORTRAN main program. See Chapter 3 of the *FORTRAN 77 Programmer's Guide* for information on writing FORTRAN programs that interact with programs written in other languages.

A main program may not be referenced from a subprogram or from itself.

1.6.4 Subprograms

A subprogram is a program unit that receives control when referenced or called by a statement in a main program or another subprogram.

A subprogram may be:

- a function subprogram identified by a FUNCTION statement
- a subroutine subprogram identified by a SUBROUTINE statement
- a block data subprogram identified by a BLOCK DATA statement
- a non-FORTRAN subprogram

Subroutines, external functions, statement functions, and intrinsic functions are collectively called procedures. A *procedure* is a program segment 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 may be written as a non-FORTRAN subprogram as described in Chapter 3 of the *FORTRAN 77 Programmer's Guide.*

1.6.5 Intrinsic Functions

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

Program units are made up of FORTRAN statements. A FORTRAN program consists of one or more program units.

1.6.6 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. Upon 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.

1.7 Executable and Nonexecutable Statements

FORTRAN statements are classified as executable or nonexecutable statements.

1.7.1 Executable Statements

An executable statement specifies an identifiable action and is part of the execution sequence in an executable program. Executable statements are organized into three classes.

Assignment statements:

- Arithmetic, logical, statement label (ASSIGN), and 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
- DO WHILE
- CALL and RETURN
- END
- END DO

Input/Output statements:

- READ, WRITE, and PRINT
- REWIND, BACKSPACE, ENDFILE, OPEN, CLOSE, and INQUIRE.
- ACCEPT, TYPE, ENCODE, DECODE, DEFINE FILE, FIND, REWRITE, DELETE, and UNLOCK.

1.7.2 Nonexecutable Statements

A nonexecutable statement is not part of the execution sequence. A statement label is permitted on most types of nonexecutable statements but that label must not be used for an executable statement in the same program unit.

A nonexecutable statement may perform one of the following 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 nonexecutable:

CHARACTER type
COMPLEX
DIMENSION
DOUBLE PRECISION
INTEGER
LOGICAL
REAL
BYTE

Other program statements that are also classified as nonexecutable include the following:

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

1.7.3 Order of Statements

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

1. 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.

2. Comment lines can be interspersed with any statement and can precede a PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statement.

3. FORMAT and ENTRY statements may be placed anywhere within a program unit after a PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statement.

4. ENTRY statements may 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

5. 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 the symbolic name of a constant must appear before the PARAMETER statement that identifies the symbolic name with that constant.

6. 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.

   The FORTRAN 77 standard is extended to permit the interspersing of DATA statements among PARAMETER statements.

   PARAMETER statements that associate a symbolic name with a constant must precede all other statements containing that symbolic name.

7. All statement function statements must precede the first executable statement.

8. IMPLICIT statements must precede all other specification statements except PARAMETER statements.

9. The last statement of a program unit must be an END statement.

Note: These rules apply to the program statements after merging of lines included by all INCLUDE statements. INCLUDE statements must precede all executable statements in order for line numbers to be correct.

1.7.4 Execution Sequence

The execution sequence in a FORTRAN program is the order in which statements are executed. The normal sequence of execution is the order in which statements appear in a program unit. This is carried out as follows:

- Execution begins with the first executable statement in a main program and continues from there.
- 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.
- Execution is returned to the calling statement via an explicit or implicit return statement.
- Normal execution proceeds from where it was suspended or from an alternate point in the calling program.
- The executable program is terminated normally when the processor executes a STOP statement in any program unit or an END statement in the main program. Execution is also terminated automatically when an operational condition prevents further processing of the program.

Normal execution sequence may 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 can cause a transfer of control are:

- GO TO
- Arithmetic IF
- RETURN
- STOP
- An input/output 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
- The terminal statement of a DO loop
- END

2. Data Types, Variables, and Arrays

2.1 Overview

In general, there are three kinds of entities that have a data type: constants, data names, and function names. FORTRAN allows these types of data:

- 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 parts, as in m + n*i*
- 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 may 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.

2.2 Data Types of Symbolic Names

A symbolic name has a definite data type in a program unit, and the type may 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 Chapter 2 of 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 to this 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 may be intermixed in expressions. The resultant value is represented in the larger of the internal representations present in the expression.

Note: The lengths of arguments in actual and formal parameter lists and COMMON blocks must agree to get predictable results.

2.3 Implicit Typing of Data

If not explicitly specified by a type statement or a FUNCTION statement, the data type of a data item or a 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 (upper case or lower case)are of type INTEGER; names beginning with all other letters are of type REAL. The default implicit data type corresponding to each letter of the alphabet may be changed or confirmed through an IMPLICIT statement.

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

2.4 Constants

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

- arithmetic constants
- logical constants
- character constants
- bit constants

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 compiler supports the following types of *arithmetic* constants: integer, real, double precision, complex, and double complex. An arithmetic constant may be signed or unsigned. A *signed constant* has a leading plus or minus sign to denote a positive or negative number, respectively. A constant that may be either signed or unsigned is an *optionally signed constant.* Only arithmetic constants can be optionally signed.

Bit constants do not have an implicit data type associated with them; their type is determined by the context in which they appear.

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

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

2.4.1 Integer Constants

An integer constant is a whole number with no decimal points; it can have a positive, negative, or zero value.

An integer constant has the following form:

sww

where:

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

An integer constant is written with a sign (optional for +) followed by a string of decimal digits interpreted as a decimal integer. When used in FORTRAN, an integer constant 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 in a following section of this chapter).
`7,909`	Embedded commas not allowed.

2.4.2 Hexadecimal Integer Constants

Hexadecimal integer constants permit the use of a base 16 radix. You must specify a dollar sign ($) as the first character, followed by any of the digits 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 the expression in which they appear.

2.4.3 Octal Integer Constants

Octal integer constants permit the use of a base 8 radix. The type of an octal integer constant is INTEGER, in contrast to the octal constant described in the section of this chapter on 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.

2.4.4 Real Constants

A real constant is a number containing a decimal point or 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 denotes 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 Chapter 2 of the *FORTRAN Programmer's Guide* for information on the magnitude and precision of a real number.

The following examples illustrate real constants written in common and scientific notation with their corresponding E format:

Common Notation	Scientific Notation	Real Exponent Form
5.0	$0.5*10$	.5E1
364.5	$3.465*10^{2}$	.3645E3
49,300	$4.93*10^{4}$	.493E5
-27,100	$-2.71*10^{4}$	-.271E5
-.0018	$-1.8*10^{-3}$	-.18E-2

Table 2-1. Notation Forms for Real Constants

The following real constants are equivalent:

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

The following table shows some invalid real constants and the reasons they are invalid.

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

Table 2-2. Invalid Real Constants

2.4.5 Double Precision Constants

A double precision constant is similar to a real constant, except it can retain more digits of the precision than a real constant. (The size and value ranges of double precision constants are given in Chapter 2 of the *FORTRAN 77 Programmer's Guide*.)

A double precision entity can assume a positive, negative, or zero value and may be written 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 used by the processor to approximate the value of the real constant. See Chapter 2 of 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
```

The following forms of the numeric value 500 are equivalent:

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

The following table shows some invalid double-precision constants and the reasons they are invalid.

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

Table 2-3. Invalid Double-Precision Constants

2.4.6 Complex Constants

A complex constant is a processor approximation to 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 for 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* + *ni*, where *i* is the square root of –1. *m* denotes the real part; *n* denotes the imaginary part. Both *m* and *n* can be positive, negative, or zero.

Examples of valid forms of complex data are:

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

Table 2-4. Valid Forms of Complex Data

The following table shows some invalid constants and the reasons they are invalid.

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

Table 2-5. Invalid Forms of Complex Data

2.4.7 Logical Constants

A logical constant represents only the values true or false.

A logical constant is specified by one of the following forms and has the indicated value:

Form	Value
`.TRUE.`	true
`.FALSE.`	false

2.4.8 Character Constants

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

Note: The quotation character (") 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 identically as a Hollerith constant.

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

The following table gives examples of valid character constants and shows how they are stored.

Constant	How Stored
`'DON''T'`	`DON'T`
`"I'M HERE!"`	`I'M HERE!`
`'STRING'`	`STRING`
`'LMN""OP'`	`LMN""OP`

Table 2-6. Valid Character Constants

The following table shows some invalid character constants and the reasons they are invalid.

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

Table 2-7. Invalid Character Constants

2.4.9 Hollerith Constants

Hollerith constants are used 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 upper or lower case) and a string of characters as specified in the character count, and has the following format:

```
nHxxx...x
```

where *n* is a nonzero, unsigned integer constant and 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
```

The following gives some examples of invalid Hollerith constants and the reasons they are invalid.

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

Table 2-8. Invalid Hollerith Constants

Rules

1. Hollerith constants are stored as byte strings; each byte is the ASCII representation of one character.

2. Hollerith constants have no type; they assume a numeric data type and size within the context in which they are used.

3. 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.

4. In other cases, when used in statements that require a specific data type, the constant is assumed to be the required type and length.

5. A length of four bytes is assumed for hexadecimal and octal constants used as arguments; no data type is assumed.

6. In other cases, the constant is assumed to be of the INTEGER*4 data type.

7. 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.

8. A variable may be defined with a Hollerith value through a DATA statement, an assignment statement, or a READ statement.

9. 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. (See Chapter 2 of the *FORTRAN 77 Programmer's Guide* for the sizes of arithmetic and logical data types.)

2.4.10 Bit Constants

Bit constants can be used anywhere a numeric constant is allowed. The allowable bit constants and their format are shown in the following table:

Format	Meaning	Valid *string* characters	Maximum
`b'` *string*`'` or `'`*string*`'b`	Binary	0, 1	64
`o'` *string*`'` or `'`*string*`'o`	Octal	0 - 7	22
`x'` *string*`'` or `'`*string*`'x`	Hexadecimal	0 - 9; a - f	16
`z'` *string*`'` or `'`*string*`'z`	Hexadecimal	0 - 9; a - f	16

*b, o, a, and z may be lower- or uppercase (B, O, X, Z)

Table 2-9. Bit Constants

Here is an example 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:

1. Bit constants have no type; they assume a numeric data type and size within the context in which they are used.

2. 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 the INTEGER*2 type and two bytes long.

3. In other cases, when used in statements that require a specific data type, the constant is assumed to be the required type and length.

4. A length of four bytes is assumed for hexadecimal and octal constants used as arguments; no data type is assumed.

5. In other cases, the constant is assumed to be of the INTEGER*4 data type.

6. A hexadecimal or octal constant can specify up to 16 bytes of data.

7. Zero padding to the left occurs when the assumed length of the constant is more than the digits specified by the constant. Truncation to left occurs when the assumed length of the constants is less that of the digits specified.

2.5 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 the execution of a program.

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 may not 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.

2.6 Character Substrings

A character substring is a contiguous sequence of *characters* hat is part of a character data item. A character substring must not be empty; i.e., 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.

2.6.1 Substring Names

A substring name allows the corresponding substring to be defined and 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.

A non-integer character can be specified *e1* and *e2*. If specified, each non-integer character is converted to integer before use; fractional portions remaining after conversion are truncated.

s is a subscript expression.

The value *e1* specifies the leftmost character position of the substring relative to the beginning of the variable or array element from which it was abstracted, while *e2* is the rightmost position. Positions are numbered left to right beginning with 1. For example, EX(3:5) denotes characters in positions three through five of the character variable EX. C(2,4)(1:5) specifies characters in positions one through five of the character array element C(2,4).

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

2.6.2 Substring Values *e1, e2*

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

$1 \leq e1 \leq e2 \leq \text{len}$

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'.
```

Examples of valid substrings taken from this variable are:

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

Table 2-10. Valid Substring Examples

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 ≥ 2 and ≤ the length of an element of BQ.
`BLT(:)`	Equivalent to the variable BLT.

2.7 Records

The record-handling capability enables you to declare and operate on multifield records in FORTRAN programs. The term *record* as it is used here is not to be confused with the term *record* used to describe input and output data records.

2.7.1 Overview of Records and Structures

A record is a composite or aggregate entity containing one ore more record elements or fields. Each element of a record can be, and usually is, 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 as arrays are.

You define the form of a record with a group of statements called a structure definition block. You 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 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. Data items with different types can be freely intermixed within a structure declaration.

Substructure Declarations

Substructures can be established within a structure by means of 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. Mapped field declarations are defined by a block of statements called a union declaration. 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

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.

2.7.2 Record and Field References

This manual uses the generic term *scalar reference* to refer 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 to structured data items defined by a RECORD statement.

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

2.7.3 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.

2.8 Arrays

An array is a nonempty 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:

- An array name
- A data type
- Array elements
- An array declarator specifying:
  - The number of dimensions
  - the size and bounds of each dimension

An array can be defined using a DIMENSION, COMMON, or type statement (described in Chapter 4); it can have a maximum of seven dimensions.

Note: For information on array handling when interacting with programs written in another language, see Chapter 3 of the *FORTRAN 77 Programmer's Guide.*

2.8.1 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 given for symbolic names. An array can be specified by the array name alone when reference is made to the array as a whole. 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.

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 types of data allowed in FORTRAN. All array elements are the same data type. The data type is specified explicitly using a type statement, or implicitly by the first character of the array name.

2.8.2 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 may appear in a DIMENSION statement, a type statement, or a COMMON statement, but not 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.

d2 is a upper dimension bound.

d1 must be a numeric expression. *d2* must be a numeric expression or an asterisk (*). An asterisk is allowed 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 is either an actual array declarator or a dummy array declarator. In an actual array declarator the array name is not a dummy argument. Conversely, a dummy array declarator is an array declarator that has a dummy argument as an array name. An array declarator may 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 must not contain a function or array element name reference.

2.8.3 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). An 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:

$(d2 - d1) + 1$

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

2.8.4 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:

1. If the actual argument is a noncharacter array name, the size of the assumed-size array equals the size of the actual argument array.

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

3. 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.

Restriction. 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 determined as described above).

2.8.5 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 leftmost subscript varies first, as compared to row order, in which the rightmost 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)
```

The elements of this array are ordered as follows:

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

2.8.6 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 may contain array element references and function references. However, it must not contain any function references that affect other subscript expressions in the same subscript.

A non-integer character can be specified for *subscript.* If specified, the non-integer character is converted to 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 2.1).

n	Dimension Declarator	Subscript	Subscript Value
1	$(j_1:k_1)$	(s_1)	$1 + (s_1 - j_1)$
2	$(j_1:k_1, j_2:k_2)$	(s_1, s_2)	$1 + (s_1 - j_1) + (s_2 - j_2)*d_1$
3	$(j_1:k_1, j_2:k_2, j_3:k_3)$	(s_1, s_2, s_3)	$1 + (s_1-j_1) + (s_2-j_2) * d_1 + (s_3-j_3) * d_2 * d_1$
n	$(j_1:k_1, \ldots j_n:k_n)$	$(s_1, \ldots s_n)$	$1 + (s_1 - j_1) + (s_2 - j_2)*d_1 + (s_3-j_3)*d_1*d_2 + \ldots + (s_n - j_n) * d_{n-1}*d_{n-2}*d_1$

Table 2-11. Determining Subscript Values

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.

3. Expressions

3.1 Overview

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

- Arithmetic
- Character
- Relational
- Logical

This section describes the formation, interpretation, and evaluation rules for each of the expressions. Mixed-mode expressions are FORTRAN 77 enhancements of FORTRAN 66 and are also discussed in this chapter.

3.2 Arithmetic Expressions

An arithmetic expression specifies a numeric computation which yields a numeric value upon evaluation. The simplest form of an arithmetic expression may be:

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

More complicated arithmetic expressions are constructed from one or more operands together with arithmetic operators and parentheses.

An arithmetic element may include logical entities because logical data is treated as integer data when used in an arithmetic context. When arithmetic and logical operands exist for a given operator, the logical operand is promoted to type INTEGER of the same byte length as the logical.

3.2.1 Arithmetic Operators

The arithmetic operators are shown in the following table:

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or Identity
-	Subtraction or Negation

Table 3-1. Arithmetic Operators

An exponentiation, division, or multiplication operator can be used only with two operands and written between the two operands. An addition or subtraction operator can be used with two operands or one operand; in the latter case, it is written preceding that operand.

Two operators may not be written 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.

3.2.2 Interpretation of Arithmetic Expressions

Interpretation of arithmetic expressions using these operators are shown below.

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

Table 3-2. Interpretation of Arithmetic Expressions

An arithmetic expression containing two or more operators is written and interpreted based on a precedence relation among the arithmetic operators, unless the order is overridden by the use of parentheses. This precedence is shown in the following table:

Operator	Precedence
( )	Highest
**	
* /	
+ –	Lowest

Table 3-3. Arithmetic Precedence

As an example:

```
    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. This produces a variation on the standard order of operations when the preceding operator is exponentiation. The unary operator must be evaluated first in that case, resulting in exponentiation taking a lower precedence in the expression.

For example:

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

Is interpreted as

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

3.2.3 Arithmetic Operands

Arithmetic operands must specify values with integer, real, double precision, complex, or double complex data types. Specific operands may be combined in an arithmetic expression. The arithmetic operands, in increasing complexity, are:

- Primary
- Factor
- Term
- Arithmetic expression

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

- Unsigned arithmetic constant
- Symbolic name of an arithmetic constant
- Arithmetic variable reference
- Arithmetic array element reference
- Arithmetic function reference
- 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 may begin with a leading plus or minus sign.

3.2.4 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:

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

In an arithmetic constant expression, the exponentiation operator is not allowed 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)
```

3.2.5 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 integer type. As with all constant expressions, no variables, array elements, or function references are allowed.

3.2.6 Rules for 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 value resulting from evaluation of a mixed-mode expression depends on the rank associated with each data type, as follows:

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)

Table 3-4. Data Type Ranks

Except for exponentiation (discussed below), the data type of the value produced by a mixed-mode expression is the data type of the highest-ranked element in the operation. The value of the lower-ranked operand is converted to the type of the higher ranked operand and the operation is performed on values with equivalent data types. For example, the data type of the value resulting from an operation on an integer operand and a real operand is real.

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

3.2.7 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 a type of the left operand.

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

x^y = EXP (y * LOG(x))

3.2.8 Integer Division

One operand of type integer may 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).

3.3 Character Expressions

A *character expression* yields a character-string value upon evaluation. The simplest form of a character expression may be:

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

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

3.3.1 Concatenate Operator

Only one character operator is defined in FORTRAN: the concatenation (//) operator. 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 is the sum of the lengths of the character operands. For example, the value of

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

is a string `'HELLO2'`.

3.3.2 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 as follows:

- 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 interpretation of the character expression

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

is a same as:

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

The value of the above character expression is the same as the constant '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.

3.3.3 Character Constant Expressions

A *character constant expression* is a character expression containing nothing that can vary. Each primary in a character constant expression must be either a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses. Variable, array element, substring, and function references are not allowed.

3.4 Relational Expressions

A relational expression yields a logical value of either .TRUE. or .FALSE. upon evaluation and comparison of two arithmetic expressions or two character expressions. A relational expression may appear only within a logical expression.

3.4.1 Relational Operators

The relational operators are:

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

The precedence among FORTRAN operators is such that arithmetic and character operators are evaluated *before* relational operators.

3.4.2 Relational Operands

The operands of a relational operator may be arithmetic or character expressions. Two operands are required to form a relational of the following form:

e1 relop e2

where:

e1 and *e2* are arithmetic or character expressions.

relop is the relational operator.

Note that *e1* and *e2* must be both arithmetic or both character.

3.4.3 Evaluating Relational Expressions

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

3.4.4 Arithmetic Relational Expressions

An arithmetic relational expression has the form:

e1 relop e2

where:

e1 and *e2*	are each an integer, real, double precision, complex, or double complex expression.
relop	is a relational operator.

Complex type operands are allowed only when the relational operator .EQ. or .NE. is used.

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. A double precision value must not be compared with a complex value.

3.4.5 Character Relational Expressions

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. It has the following form:

e1 relop e2

where *e1* and *e2* are character expressions and *relop* is a relational operator.

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. *e*1 is less than or greater than *e*2 if *e*1 is before or after *e*2 respectively in the collating sequence.

- If either *e1* or *e2* are character strings with length 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 purpose of 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. don't depend on the processor collating sequence and can be used on any processor.

3.5 Logical Expressions

A logical expression specifies a logical computation which yields a logical value upon evaluation. The simplest form of a logical expression is a:

- Logical constant
- Logical variable reference
- Logical array element reference
- Logical function reference
- Relational expression

More complicated logical expressions are constructed from one or more logical operands together with logical operators and parentheses. Five logical operators are permitted in FORTRAN and are discussed in the next section.

3.5.1 Logical Operators

The logical operators defined in FORTRAN are shown in the table below.

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

Table 3-5. Logical Operators

Only the logical negation operator .NOT. is used with one operand; all other logical operators require two operands.

When a logical expression contains two or more logical operators, the order in which the operands are combined is shown below, unless the order is changed by the use of parentheses.

Operator	Precedence
.NOT.	Highest
.AND.	•
.OR.	•
.EQV. }	
.NEQV. }	Lowest
.XOR. }	

Table 3-6. Logical Precedence

For example, in the expression

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

The operators are executed in the following sequence:

- Y .AND. Z is evaluated first. Let that result be represented as A.
- X .OR. A is evaluated second. Let this result be represented as B.
- W .NEQV.B gives the final result.

3.5.2 Logical Operands

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

- Logical primary
- Logical factor

- Logical term
- Logical disjunct
- Logical expression

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 due to the type promotion rules. After the result is computed, it may be converted back to LOGICAL if its usage requires.

Note that two logical operators may not appear in succession and that implied logical operators are not allowed.

The *logical factor* provides for the inclusion of the logical negation operator .NOT. and has the following forms:

- Logical primary
- .NOT. logical primary

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.

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.

At the highest level of complexity is the *logical expression.* A logical expression is a sequence of logical disjuncts separated by either 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 either 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 may contain arithmetic and character constant expressions but not variables, array elements, or function references.

3.5.3 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. The form and interpretation of the logical operators is defined as shown in the following table:

IF		THEN				
X1 =	X2 =	.NOT.X2	.AND.	.OR.	.EQV.	.XOR. .NEQV.
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE
.TRUE.	.FALSE.		.FALSE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.TRUE.		.TRUE.	.TRUE.	.TRUE.	.FALSE.

Table 3-7. Logical Expressions

3.6 General Rules for Evaluating Expressions

Several rules are applied to the general evaluation of 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 that 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.

3.6.1 Precedence of Operators

The precedence among arithmetic operators was given previously as:

Operator	Precedence
**	Highest
* /	Intermediate
+ -	Lowest

Table 3-8. Arithmetic Precedence

The precedence among logical operators was given previously as:

Operator	Precedence
.NOT.	Highest
.AND.	•
.OR.	•
.EQV. .NEQV. .XOR.	Lowest

Table 3-9. Logical Precedence

No precedence exists among the relational operators, and there is only one character operator, // (concatenation).

The precedence among expression operators in each type is:

Operator	Precedence
Arithmetic	Highest
Character	•
Relational	•
Logical	Lowest

Table 3-10. Precedence of Operators

3.6.2 Integrity of Parentheses and Interpretation Rules

Parentheses are used to explicitly specify the order of evaluation of operators within an expression. Expressions within parentheses are treated as an entity.

In an expression containing more than one operation, the processor first evaluates 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 or right and performs the operations according to the operator precedence described previously.

4. Specification Statements

4.1 Overview

Specification statements are nonexecutable FORTRAN statements that provide the processor information about the nature of specific data and the allocation of storage space for this data.

The specification statements are summarized below.

Statement	**Purpose**
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 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.
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. Al so, may 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	Permits a group of variables or array names to be associated 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 the above statements follow in alphabetical order.

4.2 AUTOMATIC, STATIC

Use

The STATIC keywords 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

The following table summarizes the difference between static and automatic variables upon entry and exit from a subprogram:

	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 values of the variable are retained in the static storage area.

Table 4-1. Static and Automatic Variables

AUTOMATIC variables have the following advantages:

- They permit the program to execute 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 either upon a subsequent call or return to the subprogram.

Rules for Use

1. By default, unless you specify the **–static** command line option (described in *f77*(1) in the *User's Reference Manual* and **Chapter 1** of the *FORTRAN Programmer's Guide*) all variables are AUTOMATIC *except* the following: initialized variables, common blocks and other equivalences to static data.
2. You can 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.
3. You cannot specify the AUTOMATIC or STATIC keywords in EQUIVALENCE, DATA, or SAVE statements. Any variable in these 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)
```

4.3 BLOCK DATA

Use

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

Syntax

```
    BLOCK DATA [sub]
```

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

Method of Operation

A block data subprogram is a nonexecutable program unit with a BLOCK DATA statement as its first statement, followed by a body of specification statements and terminated by an END statement. The specification statements allowed include: COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, STRUCTURE declarations, and type statements. Comment lines are permitted.

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

Rules of Use

1. The optional name *sub* is a global name and must be unique. Thus, BLOCK DATA subprograms may not have the same external name.

2. An executable program may contain more than one block data subprogram but may not contain more than one unnamed block data subprogram.

3. A single block data subprogram may initialize the entities of more than one named common block.

4.4 COMMON

Use

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]/ n list] ...
```

where *cb* is a common block name and *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 is 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 their 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, which can be accessed by all program units in which it is declared. It has no identifying name and one blank common area exists for the complete executable program.

- *Named* common storage, which has an identifying name and is accessible by all program units in which common storage with the same name is declared.

Entities in a named common may be initially defined with the DATA initialization statement in a BLOCK DATA subprogram, but entities in blank common may *not* be initialized by the DATA statement.

The number of storage units needed to store a common block is referred to as its size. This 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 need *not* be the same size in all program units.

Rules of Use

1. A variable name, array name, array declarator, or record may appear only once in all common block lists within a program unit.

2. A blank common block is specified by omitting the common block name *cb* for each list. Thus, if the first common block name is omitted, all entities appearing in the first *nlist* are specified to be in a blank common.

3. If the first *cb* is omitted, the first two slashes become optional. Two slashes with no block name between them declare the entities in the following list to be in blank common.

4. Any common block name *cb* or an omitted *cb* for blank common may 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.

5. All entities in a common block containing a character variable or character array must be of type character.

6. 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. Incorrect results are received if a common block is defined multiple times.

7. The compiler aligns entities in a common block on 32-bit boundaries. The user may change this alignment with the compiler switches **–align8**, or **–align16.** A performance degradation is associated with these switches. See the *FORTRAN 77 Programmer's Guide* for more information.

Restrictions

1. Names of dummy arguments of an external procedure in a subprogram must not appear in a common block list.

2. A variable name that is also a function name must not appear in the list.

Examples

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

The first two examples are equivalent and define a blank common block (note that these two COMMON statements must not appear in the same program unit). The third example makes the following COMMON storage assignments:

1. NAME, AGE, DRUG, and DOSE are placed in common block LABEL.

2. Y and Z are placed in blank common.

3. DOC, TIME, and TYPE are placed in common block RECORD.

The use of a COMMON statement by a subprogram and its calling program is:

```
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
```

Note that there are 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 has access to COMMON storage through 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.

4.5 DATA

Use

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).

clist is a list of the form:

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

where *a* has either of the forms:

c

r*c

- *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 rightmost characters in the constant are ignored; only the leftmost characters are stored. Each character constant initializes only one variable, array element, or substring.

Note: As an enhancement to FORTRAN 77, an arithmetic or logical entity may be initially defined using a Hollerith constant for *c* in a *clist*, using the form:

```
nHx1 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

1. 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.

2. If an unsubscripted array name is specified in *nlist*, the corresponding *clist* must contain one constant for each element of the array.

3. If two entities are associated in common storage, only one can be initialized in a DATA statement.

4. Names of entities in named common blocks may appear in *nlist* only within

a BLOCK DATA subprogram.

5. Each subscript expression in *nlist* must be an integer constant expression, except for implied-DO variables.

6. Each substring expression in *nlist* must be an integer constant expression.

7. 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.

8. 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.

9. A Hollerith constant can be used to initialize a numeric variable or array element. The rules for Hollerith assignment apply.

Restrictions

1. The list *nlist* must not contain names of dummy arguments, functions, and entities in blank common, or those associated with entities in blank common.

2. A variable, array element, or substring must not be initialized more than once in an executable program. If it is, the subsequent initializations will override the previous ones.

3. If a common block is initialized by a DATA statement in a program unit, it

cannot be initialized in other program units.

Example

```
      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 gives examples of implied-DO statements used with a 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/
```

4.6 Data Type Statements

4.6.1 Numeric Data Types

Use

Overrides implicit typing or explicitly defines the type of a constant, variable, array, external function, statement function, or dummy procedure name. Also, may specify dimensions of arrays.

Syntax

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

type is one of the keywords shown in the following table:

Type Keywords	
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	

Table 4-2. Keywords for Type Statements

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

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

Note: 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 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 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

1. Type statements are optional, and must be placed in the beginning of a program unit, but can be preceded by an IMPLICIT statement.

2. Symbolic names, including those declared in type statements, have the scope of the program unit in which they are included.

3. More than one type statement beginning with the same keyword may be included in the program unit.

4. A symbolic name must not have its type explicitly specified more than once within a program unit.

5. The name of a main program, subroutine, or block data subprogram must not appear in a type statement.

6. The compiler provides a DOUBLE COMPLEX version of the following functions:

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

Table 4- 3. Double Complex Functions

7. The *–i2* compiler option (see *f77*(1) in the *Programmer's Reference Manual* or Chapter 2 of the *FORTRAN 77 Programmer's Guide*) causes the following to take place:

   - Integer constants whose values are within the range allowed for the INTEGER*2 data types, are changed to INTEGER*2.
   - Where possible, the compiler changes to INTEGER*2 the data type of variable returned by a function.
   - Variables of the 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 example above declares that:

1. *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.

2. *total* is a real array.

3. *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.

4. *flag* and *stat* are logical variables; *bx* is a logical array.

5. *I* is a complex variable; *B* and *J* are complex arrays.

4.6.2 Character Data Types

Use

Declares the symbolic name of a constant, variable, array, external function, statement function, or dummy procedure name and specifies 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 valued integer constant expression enclosed in parentheses
- An asterisk enclosed in parentheses (*)

nam is one of the following:

v [**len*]	*v* is a variable name, symbolic name of a constant, function name, or dummy procedure name.
a [(*d*)] [**len*]	*a*(*d*) is an array declarator.

Rules for Use

1. 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.
2. A length specification immediately following an entity applies only to that entity. When an array is declared, the length specified applies to each array element.
3. If no length specification is given, a length of one is assumed.
4. 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 from 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.

5. 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 (*).

6. 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
```

In the above example:

1. *name* is a character variable of length forty.

2. *gender* has a length of one.

3. *pay* is a character array with 12 elements, each of which is 10 characters in length.

4.7 DIMENSION

Use

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 a synonym for 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

1. The dimension specification of an array can appear only once in a program unit.

2. The name of an array declared in DIMENSION statement may appear in a type statement or a COMMON statement without dimensioning information.

Examples

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

The DIMENSION statement declares *z* as an array of 25 elements, *a* as an array of 36 elements (6x6), and *ams* as an array of 50 elements (2x5x5).

4.8 EQUIVALENCE

Use

Specifies the sharing of storage units by two or more entities in a program unit thus associating those entities. This 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

An EQUIVALENCE statement specifies that the storage sequences of the entities in the list have the same first storage unit. This causes association of the entities in the list or of other elements as well. The EQUIVALENCE statement only provides association of 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 may be associated by equivalence only with other character entities. The character entities may be specified as character variables, character array names, character array element names, and 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 may cause association of other character elements as well. The lengths of the equivalenced character entities are not required to be equal.

Variables and arrays may be associated with entities in common storage. The result may be 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 of Use

1. Each subscript expression or substring expression in an equivalence list must be an integer constant expression.

2. 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.

3. An array name without a subscript is treated as an array element name that identifies the first element of the array.

4. Multidimensional array elements may 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

1. Names of dummy arguments of an external procedure in a subprogram must not appear in an equivalence list.

2. A variable name that is also a function name must not appear in the list.

3. A storage unit can appear in no more than one EQUIVALENCE storage sequence.

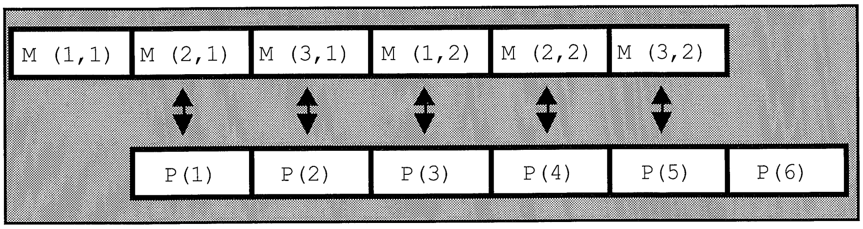
4. An EQUIVALENCE statement must not specify that consecutive storage units are to occupy nonconsecutive storage positions.

5. An EQUIVALENCE statement must not specify that a storage unit in one common block be associated with any storage unit in a *different* common block.

Example 1

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

The following figure shows the logical representation in storage caused by the above two statements:



Example 2

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

The following figure shows the logical representation in storage caused by the above two statements:

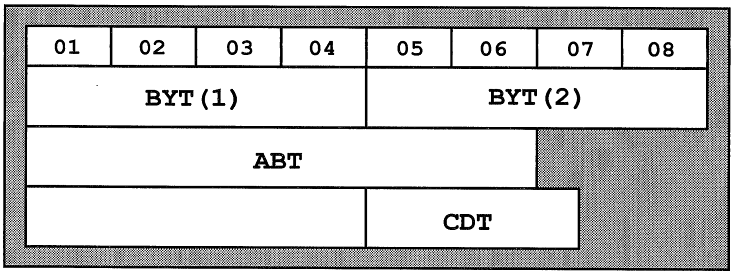


Figure 4-2. Logical Representation of EQUIVALENCE Statement

The following examples using EQUIVALENCE statements are invalid:

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

This specifies that consecutive storage units are to occupy nonconsecutive storage positions. Note that a double precision variable occupies two consecutive numeric storage units in a storage sequence.

4.9 EXTERNAL

Purpose

Identify external or dummy procedure.

Use

Specifies that a symbolic name represent an external procedure or a dummy procedure. This allows that name to be used as an actual argument in a program unit.

Syntax

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

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

Rules for Use

1. 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.

2. 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.

3. A symbolic name must appear only once in all of the EXTERNAL statements of a program unit.

4. 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.

Restrictions

A statement function name must not appear 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.

4.10 IMPLICIT

4.10.1 Use

Changes or defines default implicit type of names.

4.10.2 Syntax 1

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

where:

type is one of the keywords shown in the following table:

Type Keywords

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

Table 4- 4. Keywords for Type Statements

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.

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 valued integer constant expression enclosed in parentheses

If *len* is not specified, the value of *len* is 1.

Method of Operation – Syntax 1

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. The IMPLICIT statement can be used 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.

4.10.3 Syntax 2

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

Method of Operation – Syntax 2

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.

4.10.4 Syntax 3

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

Note: UNDEFINED and NONE are synonyms; each performs the same function, which is described in the next section.

Method of Operation – Syntax 3

When a type isn't 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.

The compiler permits you, using either the `implicit undefined` or `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 example

```
    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.

4.10.5 Rules for Use – All Syntaxes

1. IMPLICIT statements must precede all other specification statements except PARAMETER statements.

2. Multiple IMPLICIT statements are allowed in a program unit.

3. IMPLICIT statements cannot be used to change the type of a letter more than once inside a program unit. Since letters can be part of a range of letters as well as single, ranges of letters must not overlap.

4. Lower- and upper-case alphabetic characters are *not* distinguished. Implicit type is established for *both* the lower- and upper-case alphabetic characters or range of alphabetic characters regardless of the case of *l*1 and *l*2.

5. The **-u** command line option turns off all default data typing and any data typing explicitly specified by an IMPLICIT statement.

4.10.6 Examples

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

The example above declares that:

1. All variables with names beginning with the letters F, M, N, O, P, f, m, n, o, or p are of type INTEGER and are assigned the STATIC attribute.

2. All variables with names beginning with the letter b or d are of type REAL, except for variables bin and dale only, 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)
```

4.11 INTRINSIC

Purpose

Identify intrinsic function or system subroutine.

Use

Identifies a symbolic name as being the name of an intrinsic function or a system subroutine. 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

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

Restrictions

1. A name may not appear in both INTRINSIC and EXTERNAL statements in the same program unit.
2. A name must appear only once in all of the INTRINSIC statements of a program unit.

3. The names of intrinsic functions which 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.

Examples

Consider the following statements:

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

and its 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.

4.12 NAMELIST

Use

Permits a group of variables or array names to be associated 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 of Use

1. The items in *namelist* are read or written in the order they are specified in the list.

2. The items can be of any data type, which can be specified either explicitly or implicitly.

3. The following items are not permitted in *namelist*:

   - Dummy arguments

   - Array elements, character substrings, records, and record fields

See also the description of the READ and WRITE statements in Chapter 8 for more information on *namelist* directed I/O.

Examples

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

In the above example, *input*, when specified to namelist-directed I/O statement, refers to *item* and *quantity*; likewise, *output* refers to *item* and *total*.

4.13 PARAMETER

Use

Gives a constant a symbolic name.

Syntax

Format 1

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

Format 2

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

where *p* is a symbolic name and *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 to 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 of the 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 or
- 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 this chapter for details.

Format 2, which has no bounding parentheses, causes the symbolic name to be typed by the form of the actual constant which 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

1. If *p* is of type integer, real, double precision, or complex, then *e* must be an arithmetic constant expression.

2. If *p* is of type character or logical, then *e* must be a character constant expression or a logical constant expression, respectively.

3. 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.

4. A symbolic name of a constant must be defined only once in a PARAMETER statement within a program unit.

5. The data type of a named constant must be specified by a type statement or IMPLICIT statement prior to its first appearance in a PARAMETER statement, if a default implied type is not to be assumed for that symbolic name.

6. Character symbolic named constants must be specified as character type 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 prior to the definition of the name in the PARAMETER statement.

7. Once a symbolic name is defined, it can be used as a primary in any subsequent expressions or DATA statements in that program unit.

8. The functions IAND, IOR, NOT, IEOR, ISHFT, LGE, LGT, LLE, and LLT with constant operands can be specified in a logical expression.

9. The function CHAR with a constant operand can be specified in a character expression.

10. The functions MIN, MAX, ABS, MOD, ICHAR, NINT, DIM, DPROD, CMPLX, CONJG, and IMAG with constant operands can be specified in arithmetic expressions.

11. Symbolic names cannot be used to specify the character count for Hollerith constants.

12. Symbolic constants can appear in a FORMAT statement only within the context of a general expression bounded by angled brackets: <>.

13. 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 instance, 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

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

The example above declares that 1 is converted to 1E0, making X the name of a REAL constant.

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

3.14 is converted to 3, making I the name of an INTEGER constant.

In the following example, *interest_rate* is assigned the constant value of .087769.

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

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

```
PARAMETER interest_rate = .087769
```

4.14 POINTER

Use

Establishes pairs of variables and pointers, where each pointer contains the address of its paired variable.

Syntax

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

where *v1*, *v2* are pointer-based variables, and *p1*, *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

1. Once you have defined a variable as based on a pointer, you must assign an address to that pointer. You then 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.) Is is your responsibility to provide an address of a variable of the appropriate type and size.

2. 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 since no storage is allocated when a pointer-based variable is defined.

Restrictions

1. A pointer-based variable cannot be used as a dummy argument or in COMMON, EQUIVALENCE, DATA, or NAMELIST statements.

2. A pointer-based variable cannot itself be a pointer.

3. The dimension expressions for pointer-based variables must be constant expression in main programs. In subroutines and function, the same rules apply for pointer-based variable as for dummy arguments. The expression can contain dummy arguments and variable in COMMON. Any variable in the expressions must be defined with an integer value at the time the subroutine or function is called.

Example

```
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
```

4.15 PROGRAM

Use

Defines a symbolic name for the main program.

Syntax

```
    PROGRAM pgm
```

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

Rules of Use

1. The PROGRAM statement is optional. However, it must be the first statement in the main program when used.

2. 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.

4.16 RECORD

Use

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 this chapter) and record-name is a variable, an array, or an array declarator.

Method of Operation

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

Examples

```
STRUCTURE /WEATHER/
    INTEGER        MONTH, DAY, YEAR
    CHARACTER*40   CLOUDS
    REAL                   RAINFALL
END STRUCTURE
RECORD /WEATHER/ LATEST, PAST (1000)
```

The record *latest* has the format specified by the structure *weather*; *past* is an array of 1000 records, each record having the format of the structure *weather*.

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 1000 specifying the target array element. See the description of the STRUCTURE statement for an example of declaring a structure format.

4.17 SAVE

Use

Retains the values of variables and arrays after execution of a RETURN or END statement in a subprogram. This allows those entities to 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 upon exit from a subprogram, except in the following cases:

- Entities specified by a SAVE statement
- Entities in blank common
- Entities 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 upon exit from the subprogram.

Rules of Use

1. A SAVE statement without a list is treated as though all allowable entities from that program unit were specified on the list.
2. A SAVE statement may be placed in the main program but it has no effect.
3. A given symbolic name may 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

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

4.18 STRUCTURE / UNION

Use

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	is a name used to identify 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 may 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 on the next page.
- PARAMETER statements, which do not affect the form of the structure.

UNION Declaration Syntax

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

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

Method of Operation

1. 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.

2. 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.

3. 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 above example, the STRUCTURE statement produces the following storage mapping for the *latest* specification in the RECORD statement:

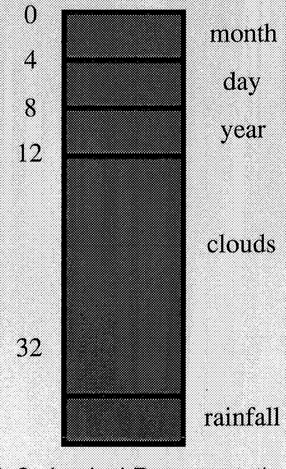


Figure 4–3. Logical Representation of STRUCTURE Statement

The following gives an example of initializing the fields of a 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
```

This example causes the following text to be written to the standard input/output device:

```
    081089
```

4.19 VOLATILE

Use

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. When two or more names are specified, each of the volatile-items must be separated by a comma.

For more information on optimization, refer to the *Languages Programmer's Guide* and the *f77*(1) manual page in the *User's Reference Manual.*

5. Assignment and Data Statements

5.1 Overview

Assignment statements assign values to variables and array elements. Five types of assignment statements are included in FORTRAN:

- Arithmetic assignment
- Logical assignment
- Character assignment
- Statement label assignment

DATA statements, implied DO lists in DATA statements, and BLOCK DATA subprograms are closely related and can be used to initialize variables and array elements. BLOCK DATA subprograms are described in detail in Chapter 4.

5.2 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 a 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.

Both *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. The type conversion rules are:

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

Table 5–1 gives the detailed conversion rules for arithmetic assignment statements.

The functions in the second column of the table are intrinsic functions described in Chapter 10 and Appendix A.

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

Variable or Array Element (v)	Expression (e)		
	INTEGER or LOGICAL	REAL	REAL*8
INTEGER or LOGICAL	Assign e to v	Truncate e to integer and assign to v	Truncate 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
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
REAL*16	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
COMPLEX*16	Append fraction to e and assign to v; imaginary part of v is 0.0	Assign e to high order portion of real part of v; imaginary part of v is 0.0	Assign e to real part of v; imaginary part is 0.0

Table 5–1. Conversion Rules for Assignment Statements

Variable or Array Element (v)	Expression (e)		
	REAL*16	COMPLEX	COMPLEX*16
INTEGER or LOGICAL	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	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	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
COMPLEX	Assign high order portion of e to real part of v; low order part is rounded; imag part of v is 0.0	Assign e to v	high order parts of real & imaginary components of e are assigned to v; low order parts are rounded
COMPLEX*16	As above	Assign e to high order parts of v; low order parts of v are 0.	Assign e to v

Table 5–1 (continued). Conversion Rules for Assignment Statements

5.3 Logical Assignment Statements

The logical assignment statement is used to assign the value of a logical expression to a logical variable or array element. It takes the form:

$v = e$

where *v* is a name of a logical variable or logical array element and *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.

5.4 Character Assignment

The character assignment statement is used to assign 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 a name of a character variable, array element, or substring and *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.

The entity *v* and character expression *e* may 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 examples show character assignment:

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

If 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.

5.5 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 for more information.

5.6 ASSIGN

The ASSIGN statement causes a statement label to be assigned to an integer variable and is used in conjunction with an assigned GOTO statement or an input/output 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.

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 input/output 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 may be redefined with the same statement label, a different statement label, or as 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
```

5.7 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.

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

5.8 Implied-DO

5.8.1 Use

Initializes or assigns initial values to elements of an array.

5.8.2 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.

5.8.3 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 a implied-DO list, see the DATA section in Chapter 4.

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

5.8.4 Rules

The integer constant expressions used for *e1*, *e2*, and *e3* may 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 may contain implied-DO variables of implied-DO lists that have the subscript expression within their range.

6. Control Statements

6.1 Overview

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

Command	Purpose
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.

Table 6-1. Control Statements

Command	Purpose
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 may appear only in a function or subroutine program.
STOP	Allows termination of an executing program.

Table 6-1. (continued) Control Statements

6.2 CALL

Use

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

Execution of a CALL statement causes an evaluation of 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 of Use

1. 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.

2. A subroutine that has been defined without an argument may be referenced by a CALL statement of the following forms:

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

3. 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.

4. 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.

5. 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 is another exception to rule (1) above.

6. A subroutine can call itself directly or indirectly (recursion).

Note: Recursion is an extension to FORTRAN 77. FORTRAN 77 does not permit 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
*
```

6.3 CONTINUE

Use

Has no operational function; 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 the subhead Loop control processing under the DO statement 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
```

6.4 DO

Use

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 optionally be omitted. If *s* is omitted, the loop must be terminated with an END DO statement. Upon 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 DO 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, upon encountering a DO loop whose upper or lower limit has already been reached, the compiler-generated code skips the loop. This 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 done by specifying the **–onetrip** compile 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 of Use

1. DO-loops can be nested but must not overlap.

2. 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.

3. 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.

4. The same statement may serve as the terminal statement in two or more nested DO-loops.

5. The terminal statement for two or more nested DO-loops can only be branched to by a GOTO statement in the innermost loop. If one of the outside loops has a GOTO statement with branches to this terminal statement, the result is unpredictable.

Restrictions

1. The following statements must not be used 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	

2. If the statement labeled *s* is a logical IF statement, then it can contain any executable statement in its statement body, except for:

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

3. Except by the incremental process covered above, the DO variable must not be redefined during execution of the range of the DO-loop.

4. 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.

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 10 times, then execution resumes at the statement (E) following CONTINUE.

6.5 DO WHILE

Use

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 prior to 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*. I f *e* is omitted, execution resumes at the first statement following the END DO statement.

6.6 ELSE

Use

Used in conjunction with the block IF or ELSE IF statements.

Syntax

```
    ELSE
```

Method of Operation

Two terms need to be defined to explain the ELSE statement:

> ELSE-block (defined below) and IF-level (defined on the block IF statement page).

An *ELSE-block* is the code that is executed when an 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. As well as containing simple, executable statements, an ELSE-block may be empty (contain no statements) or may contain embedded block IF statements. Don't confuse the ELSE-block and 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 has no logical expression to evaluate; 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 of Use

1. There cannot be any ELSE IF or ELSE statements inside an ELSE-block at the same IF-level.

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

Restrictions

1. The ELSE-block can be entered only by executing the ELSE statement. No transfer of control that jumps from outside the ELSE-block into the ELSE-block is allowed.

2. If an ELSE statement has a statement label, the label cannot be referenced by any statement.

Example

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

6.7 ELSE IF

Use

Used optionally with the block IF statement.

Syntax

```
ELSE IF (e) THEN
```

where *e* is a logical expression.

Example

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

6.8 END

Use

Indicates 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 of Use

1. An END statement must be the last statement in every program unit.

2. An END statement must not be continued. No other statements should have END as the first three non-blank characters.

6.9 END DO

Use

Defines the end of a indexed DO loop or a DO WHILE loop.

Syntax

```
    END DO
```

6.10 END IF

Use

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

Syntax

```
    END IF
```

Rules of Use

1. Every block IF statement requires an END IF statement to close that IF-level. (IF-level is described under the block IF statement.)

2. The IF-level of an END IF statement must be greater than zero (there must be a preceding corresponding block-IF statement).

Example

See the example given with the description of the ELSE statement.

6.11 GO TO (Unconditional)

Use

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

```
GO TO 358
```

Program control is transferred to statement 358 and normal sequential execution continues from there.

6.12 GO TO (Computed)

Use

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, and *i* is an integer.

A non-integer expression may also be used for *i*. Non-integer expressions are converted to integer (fractional portions are discarded) before being used to index into the list of statement labels.

Method of Operation

A computed GO TO statement causes evaluation of the integer expression, followed by transfer of control.

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.

Rules of Use

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

Example

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

Program control is transferred to statement 350 (KVAL + 1).

6.13 GO TO (Symbolic Name)

Use

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.

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 of Use

1. The same statement label may appear more than once in the same assigned GO TO statement.

2. If the 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.

6.14 IF (Arithmetic)

Use

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 statement 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 of Use

The same statement number may be used more than once in the same arithmetic IF statement.

Example

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

- If the expression evaluates to be negative, control jumps to statement 500.
- If the expression evaluates to be zero, control jumps to statement 1000.
- If the expression evaluates to positive, control jumps to statement 1500.

6.15 IF (Branch Logical)

Use

Allows conditional statement execution.

Syntax

```
IF (e) st
```

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

Method of Operation

Execution of a logical IF statement causes 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 may affect entities in the statement *st*.

Example

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

6.16 IF (Test Conditional)

Use

Allows 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

Two terms need to be defined to explain the block IF statement: IF-block and IF-level.

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 may be empty (contain no statements) or may contain embedded block IF statements. Don't confuse IF-block with block IF.

Block IF statements and ELSE IF statements may be embedded, which can make figuring which statements are in which conditional blocks very 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 *s* is:

```
    (n1 - n2)
```

where (starting count at the beginning of the program unit): *in1* 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 IF-block).

When a block IF statement is reached, the logical expression *e* is evaluated. If *e* evaluates to 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* evaluates to 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 IF statement at the same IF-level as the block IF statement.

Restrictions

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

Example

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

6.17 PAUSE

Use

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 input
will terminate job.
```

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

```
Pause "Console Check"
```

causes the following message to be displayed:

```
PAUSE Console Check statement executed
To resume execution, type go.  Any other input
will 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.

6.18 RETURN

Use

Returns control to the referencing program unit. It may 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 non-integer expression may be used for *e*. Non-integer 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 the following:

- Entities that are specified in a SAVE statement
- Entities that are in blank or named common
- Entities that are initialized in a DATA statement that have neither been redefined nor become undefined

6.19 STOP

Use

Allows termination of 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 supplied, the digit string or character constant becomes accessible to the system as program termination status information.

7. Input/Output Processing

7.1 Overview

Input statements copy data from external media or from an internal file to internal storage. This is called *reading*. Output statements copy data from internal storage to external media or to an internal file. This is called *writing*.

Input/output facilities in FORTRAN give you control over the input/output system. This section deals primarily with the programmer-related aspects of input/output processing rather than with the implementation of the processor-dependent input/output specifications.

See Chapter 1 of the *FORTRAN 77 Programmer's Guide* for information on extensions to FORTRAN 77 that affect input/output processing.

7.2 Records

A *record* is simply a sequence of values or characters. FORTRAN has three kinds of records: formatted, unformatted, and endfile records. A record is a logical concept; it does not have to correspond to a particular physical storage form. However, external media limitations may also limit the allowable length of records.

7.2.1 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 input/output 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.

7.2.2 Unformatted Records

Unformatted records contain sequences of values, both character and non-character, 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.

An unformatted record can be read from or written to only by unformatted input/output 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 may be empty.

7.2.3 Endfile Records

An *endfile record* marks the logical end of a data file. Thus, it may occur only as 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.

7.3 I/O Statements

The input/output 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 input/output.

7.3.1 Unformatted Statements

An unformatted input/output statement transfers data in the non-character format during an input/output operation. Unformatted input/output operations are usually faster than formatted operations, which translate data to character format.

In processing formatted statements, the system interprets some characters—for example, the line-feed character—as special controls and eliminates them from read records. Therefore, unformatted statements must be used when *all* characters in a record are required.

The absence of a format specifier (as well as an asterisk in the 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.

7.3.2 Formatted Statements

A *formatted* input/output 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 reference a format statement with an *a* character descriptor (the descriptors are described in detail in **Chapter 9**); both statements perform the same function, namely writing the message *Hello World* to the device associated with Unit 6.

7.3.3 List-Directed Statements

A *list-directed* input/output 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.

```
          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 make this determination. Note that list-directed statements contain an asterisk as the format specifier rather than a reference label.

7.4 Files

A file is a sequence of records. The processor determines the set of files that *exist* for each executable program. The set of existing files can vary while the program executes. Files that are known to the processor do **not** necessarily exist for an executable program at a given time. A file may exist and contain no records (all files are empty when they are created). Input/output statements can only be applied to files that exist.

Files may have names; if they do, they are called *named files.* Names are simply character strings.

Every data file has a position. The position is used by input/output 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.

There are two kinds of files: internal files and external files.

7.4.1 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; that is, it can contain no records.

7.4.2 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 prior to data transfer. Reading and writing records is only by sequential access formatted input/output statements that do not specify list-directed formatting.

The following is a simple example showing the use of 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
        WRITE (NUMERICALREP, 999) CHARREP
999     FORMAT (A3)
        END
```

In the first example, the contents of *NumericalRep* is converted to character format and placed in *CharRep*; in the second example, the contents of *CharRep* is converted to integer and placed in *NumericalRep.*

7.5 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 file must be accessed sequentially.

As an extension, the use of internal files in both formatted and unformatted input/output operations is permitted.

7.5.1 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 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, etc.
- Formatted and unformatted records cannot be mixed in a file.
- The last record of the file may be an endfile record.
- The records of a pure sequential file must not be read or written by direct access input/output statements.

7.5.2 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 is 1, 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.
- Only direct access input/output statements may be used for reading and writing records. List-directed formatting is not permitted.
- 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.

7.5.3 Keyed Access

The *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 field(s) are

defined when the file is created with an OPENstatement. Each READ statement contains a key to locate the desired record in the indexed file.

Use of the keyed access method and the sequential access method on the same opened file can be intermixed.

7.6 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 may not be connected to more than one unit or vice-versa. Each program has a processor dependent set of existing units. A unit has two states: connected and disconnected.

7.6.1 Connection of a Unit

A *connected unit* refers to a data file. A unit can be implicitly connected 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 may be connected and not exist. Consider, for example, a unit preconnected to a new file. A *preconnected unit* is a unit that 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.

As stated above, a file may only be connected to one unit and a unit may only be connected to one file.

7.6.2 Disconnection of a Unit

A unit may be *disconnected* from a file by a CLOSE statement specifying that particular unit.

8. Input/Output Statements

This chapter describes the statements that control the transfer of data within internal storage, and between internal storage and external storage devices. This chapter has the following major sections:

- **Input/Output Statement Summary**, an overview of the input/output statements described in this chapter.
- **Statement Descriptions,** which give the syntax, rules, and examples for each input/output statement.
- **Control Information List** and Input/Output List, which describe in detail parameters specified as part of an input/output statement. The lists specify data source and *target* locations, data format, and other information associated with data transfer.
- **Data Transfer Rules**, which give general rules that apply to data transfer statements.

8.1 Statement Summary

The input/output statements described in this chapter are grouped into the following classes:

1. Data transfer statements, which transfer information between two areas of internal storage or between internal storage and an external file. They are:
   - READ
   - DELETE
   - UNLOCK

- ACCEPT
- WRITE
- REWRITE
- PRINT or TYPE

2. Auxiliary statements, which explicitly open or close a file, provide current status information about a file or unit, or write an endfile record. They are:

   - OPEN
   - CLOSE
   - INQUIRE
   - ENDFILE

3. File positioning statements, which position data files to the previous record or to the file's initial point. These statements apply only to external files and they are:

   - BACKSPACE
   - REWIND

4. 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

8.2 ACCEPT

Purpose

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 and *iolist* is an optional output list specifying where the data is to be stored. See the Control Information List and Input/Output List sections of this chapter for a description of the *f* and *iolist* parameters.

Rules of 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 the Data Transfer Rules section of this chapter for additional rules.

Example

```
          ACCEPT 3,x
     3    FORMAT (A)
```

transfers character data from the standard input unit into X.

8.3 BACKSPACE

Purpose

Positions a data file before the preceding record. It may be used with both formatted and unformatted data files.

Syntax

```
BACKSPACE u
BACKSPACE (alist)
```

where *u* is an external unit identifier and *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 *input/output 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 to be 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 with the an ACCESS="KEYED" or ACCESS="APPEND" 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 (exception: see *unit specifier*).

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)
```

8.4 CLOSE

Purpose

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= is omitted, then *u* must be the first specifier in *cilist*.
IOSTAT=*ios*	is an *input/output 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.

DISPOSE=*disposition*
DISP=*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 which, 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 may be reconnected within the same executable program, either to the same file or to a different file. A file that is disconnected may 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')
```

8.5 DECODE

Purpose

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, the concept of multiple records isn't supported, and the record length is specified by the user. Where possible, use a READ statement instead of DECODE in new programs to make them compatible among different FORTRAN 77 operating environments.

Syntax

```
    DECODE (n,f,target[,ERR=s][,IOSTAT= rn]) [iolist]
```

Parameter Explanations

n	An integer expression specifying the number of characters to be translated to internal format.
f	Format specifier (as described in the the section on Format Specifier in this chapter).

ERR=*s*
IOSTAT=*rn*
	See the section on the Control Information List in this chapter for an explanation of these parameters.

target	Scalar reference or array indicating the destination of the characters after translation to external form.
iolist	Optional list specifying the source data, as described in the Input/Output List section of this chapter.

Method of Operation

1. The relationship between the I/O list and the format specifier is the same as for formatted I/O.

2. 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.

8.6 DEFINE FILE

Purpose

Defines the size and structure of a relative file and connects it to a unit. Provides primarily the same function as the FORTRAN OPEN statement specifying `ACCESS='DIRECT'`.

Syntax

```
    DEFINE FILE u (reccount,reclen,U,a svar)
                 [,u (reccount, reclen, U, asvar)] …
```

Parameter Explanations

u	Integer expression that identifies the number of an external unit that contains the file
reccount	Integer expression defining the number of records in the file.
reclen	Integer expression specifying in word (two-byte) units the length of each record.

U	Specifies an unformatted (binary) file. The U entry is always required and always in the same position shown in the above syntax.
asvar	Associated variable; *asvar* is an 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

1. Only formatted files can be opened with a DEFINE FILE.

2. The file defined by *u* is assumed to contain fixed-length records of two bytes each.

3. The records in the file are numbered 1 through *reccount.*

4. The DEFINE FILE statement or equivalent OPEN statement must be executed before executing a READ, WRITE, or other direct access statement.

5. The first direct access READ for the specified file opens an existing file; if the file doesn't exist, an error condition occurs.

6. The first direct access WRITE for the specified file opens the file, and creates a new relative file.

8.7 DELETE

Purpose

Deletes a record from an indexed file. An error condition occurs if the file is not indexed.

Syntax

```
    DELETE [UNIT=] unum
    DELETE ([UNIT=] unum[,IOSTAT=rn][,ERR=s]
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.

See **Control Information List** and **Input/Output List** sections in this chapter for details on these parameters.

Method of Operation

Deletes the current record, which is the last record accessed on unit *u* .

Examples

```
    DELETE (10)
```

deletes the last record read in from the file connected to logical unit 10.

8.8 ENCODE

Purpose

Transfers data between internal files, encoding the transferred data from internal format to character format.

Note: This statement provides primarily 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 isn't supported, and the record length is specified by the user. Where possible, use a WRITE statement instead of ENCODE in new programs to make them compatible among different FORTRAN 77 operating environments.

Syntax

```
    ENCODE (n,f,target[,ERR=s][,IOSTAT= rn]) [iolist]
```

Parameter Explanations

n	An integer expression specifying the number of characters to be translated to character format.
f	Format specifier (as described in the Format Specifier section of this chapter).
ERR=*s*	See the Control Information List section in this chapter for an
IOSTAT=*rn*	explanation of these parameters.
target	Scalar reference or array indicating the destination of the characters after translation to external form.
iolist	Optional list specifying the source data, as described in the Input/Output List section of this chapter.

Method of Operation

1. The relationship between the I/O list and the format specifier is the same as for formatted I/O.
2. The *target* is padded with blanks if fewer than *n* characters are transferred.
3. 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.

8.9 ENDFILE

Purpose

Writes an endfile record as the next record of the file. It may be used with both unformatted and formatted data files.

Syntax

```
ENDFILE u
ENDFILE (alist)
```

where *u* is an external unit identifier and *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 *input/output 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 to be 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 (exception: see *unit specifier*).

An ENDFILE statement writes an endfile record and 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 prior to the execution of any data transfer input/output statement.

Note: However, if the program is compiled with the **–vms_endfile** option, the file can still be written to after the endfile record.

Examples

```
ENDFILE 2
ENDFILE (2,IOSTAT=IE, ERR=1000)
```

8.10 FIND

Purpose

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. Functionally equivalent to a direct access READ statement except 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 ])
```

The parameter list is as follows:

u	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 the Control Information List section in this chapter for an explanation of these parameters.

8.11 INQUIRE

Purpose

Inquires about the properties of a particular named file or of the file connected to a particular unit. There are two forms: inquire by file and inquire by unit.

Syntax

```
INQUIRE (FILE=fname [DEFAULTFILE= name ...]
inqlist
INQUIRE (UNIT=] u, inqlist
```

FILE=*fname*	is a *file specifier*. *fname* is one of the following: a character expression, that specifies the name of the file being queried. The named file need not exist or be connected to a unit.
[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.*

DEFAULTFILE=*dname*

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 the description of the OPEN statement for details.

inqlist is composed of one or more of the following parameters:

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 follows:

Value Assigned	File Access
`SEQUENTIAL`	Sequential
`DIRECT`	Direct
`KEYED`	Keyed
`UNKNOWN`	No Connection

Table 8-1. File Access Types

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 follows:

blnk=	**Specifier**
NULL	Null blank control, connected for formatted input/output
ZERO	Zero blank control
UNKNOWN	Not connected, or not connected for formatted input/output

Table 8-2. Blank Control Specifiers

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 an allowed access method for the file.

dir is assigned the value NO if DIRECT is not an allowed 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 to be 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 follows:

fm=	**Specifier**
FORMATTED	Formatted input/output
UNFORMATTED	Unformatted input/output
UNKNOWN	Unit is not connected

Table 8-3. Form Specifiers

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 an allowed form for the file. *fmt* is assigned the value NO if FORMATTED is not an allowed form. If the processor is unable to determine the allowed forms of data transfer, *fmt* is assigned the value UNKNOWN.

IOSTAT=*ios* is an *input/output 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 follows:

keystat=	**Meaning**
YES	Indexed file, keyed access allowed
NO	Keyed access not allowed
UNKNOWN	Access type undetermined

Table 8-4. Keyed Access Status Specifiers

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 must not 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 or in processor-dependent units for formatted or unformatted input/output, respectively. 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 an allowed access method for the file. *seq* is assigned the value NO if SEQUENTIAL is not an allowed access method. If the processor is unable to determine the allowed 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 an allowed format for the file.

unf is assigned the value NO if UNFORMATTED is not an allowed format for the file. If the processor is unable to determine the allowed form, *unf* is assigned the value UNKNOWN.

Method of Operation

Specifiers can be given in *iflist* or *iulist* in any order (exception: see *unit specifier*).

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

```
    INQUIRE (FILE='MYFILE.DATA',NUMBER=IU,RECL=IR)
    INQUIRE (UNIT=6, NAME=FNAME)
```

8.12 OPEN

Purpose

OPEN 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 *input/output 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 file name. Three VMS predefined system logical names— SYS$INPUT, SYS$OUTPUT, and SYS$ERROR— are supported. These 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 which, when trailing blanks are removed, has either of the 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, then *iolist* must also contain a record length specifier. If *iolist* contains no access specifier, the value SEQUENTIAL is assumed.

KEYED specifies that the file is to be accessed by a key-field value.

APPEND specifies sequential access that, after execution of an OPEN statement, the file is to be positioned after the last record.

ASSOCIATEVARIABLE=*asvar*
Direct access only. After each input/output 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 which, when all trailing blanks are removed, has the value NULL (the default) or ZERO.

NULL specifies that blank characters in numeric formatted input fields are to be ignored.

ZERO specifies that all blanks other than leading blanks are to be treated as zeros. If *iolist* contains no blank specifier, the value NULL is assumed.

CARRIAGECONTROL=*type*
type is a character expression that determines carriage control processing as follows:

type=	Meaning
`FORTRAN`	Standard FORTRAN interpretation of the first character
`LIST`	Single spacing between lines
`NONE`	No implied carriage control

Table 8-5. Carriage Control Options

LIST is the default for formatted files and NONE is the default for unformatted files. When the **—vms_cc** option (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 pathname or an alternate prefix filename for the opened unit. When specified, the full file name 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 file name.

DISPOSE=*disposition*
DISP=*disposition*

disposition is a character expression that designates how the opened file is to be handled after it is closed, the table below gives the values that can be specified for *disposition* and the effect on the closed file.

DISPOSE=	**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.

Table 8-6. Disposition Options

FORM=*fm* is a *form specifier*. *fm* is a character expression which, when all trailing blanks are removed has either of the values FORMATTED or UNFORMATTED. The file is connected for formatted or unformatted input/output, respectively. If *iolist* contains no form specifier, the default value FORMATTED is assumed for sequential and direct access file.

As an extension, BINARY can also be used to specify the form of the file. This 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.

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 120 bytes.
- *key1start* and *key1end* are integers defining the start and ending byte position of the primary field, which is always required. *key2start* and *key2end*, and subsequent specifications, define the starting and ending position 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 the Control Information List section of this chapter. 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 this keyword isn't specified, both reading and writing to the specified file is is permitted.
RECORDSIZE=*rl*	Same as RECL.

RECORDTYPE=*rt*

where *rt*, when creating a file, 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 as follows:

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 the table shown above. • 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 doesn't specify a full record.
SHARED	Specifies that other programs have concurrent access to the file.
STATUS=*sta*	is a *file status specifier*. *sta* is a character expression which, ignoring trailing blanks, has one of the following values: OLD. The FILE=*fname* specifier must also be given and the file must exist. NEW. The FILE=*fname* specifier must be given; the file is created by open and the file status automatically turned to OLD. A file with the same name must not already exist. SCRATCH. An unnamed file will be created which is connected to the unit from [UNIT=] and will be deleted when that unit is closed by CLOSE. Named files should not be used with SCRATCH. UNKNOWN. Meaning is processor dependent. See the Unknown Status section in Chapter 1 of the *FORTRAN 77 Programmer's Guide* for more information. If the STATUS parameter is omitted, UNKNOWN is the default.
TYPE=*sta*	Same as STATUS.

Rules of Use

1. Specifiers may be given in *iolist* in any order (exception: see *unit specifier*).

2. The unit specifier is required; all other specifiers are optional. The record length specifier is required for connecting to a direct access file.

3. The unit specified must exist.

4. 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.

5. 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.

6. 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 for the value of the BLANK specifier.

7. See the Data Transfer Rules section of this chapter for additional rules.

Examples

```
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))
```

8.13 PRINT or TYPE

Purpose

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 the **Control Information List** and **Input/ Output List** sections of this chapter.

TYPE is a synonym for and can be used instead of PRINT.

Rules of Use

The PRINT statement may be used for formatted output to the system output unit. See the **Data Transfer Rules** section of this chapter for additional rules.

Examples

```
    PRINT 10, (FORM (L), L=1,K+1)
    PRINT *, X,Y,Z
    TYPE *, ' VOLUME IS ',V,' RADIUS IS ',R
```

8.14 READ (Direct Access)

Purpose

Transfers data from an external file to the items specified by the input list. Transfer occurs using the direct access method (see Chapter 7).

Syntax

Formatted

```
 READ ([UNIT=]unum, REC=rn, f [,IOSTAT=ios]
+     [,ERR= s]) [iolist]
```

Unformatted

```
 READ ([UNIT=]unum, REC=rn, [,IOSTAT=rn]
+     [,ERR=s ]) [iolist]
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
f	A format specifier
REC=*rn*	Direct access mode. The number of the record to be accessed.
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=s	Statement label to which control is transferred after an error.
iolist	Specifies memory location where data is to be read.

See Control Information List and Input/Output List sections in this chapter for details on these parameters.

See the Data Transfer Rules section of this chapter and Chapter 7 for more information on formatted and unformatted input/output.

8.15 READ (Indexed)

Purpose

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).

Syntax

Formatted

```
 READ ([UNIT=]unum,f,KEY=val[,KEYID=kn]
+   [,IOSTAT=rn][,ERR=s][) [iolist]
```

Unformatted

```
 READ ([UNIT=]unum,key[,keyid][,IOSTAT=rn]
+   [,ERR=s] ) [iolist]
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
f	A format specifier.
KEY=*val*	Value of key field in record to be accessed.
KEYID=*kn*	Key-reference specifier.
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.
iolist	Specifies memory location where data is to be read.

See Control Information List and Input/Output List sections in this chapter for details on these parameters.

See the Data Transfer Rules section of this chapter and Chapter 7 for more information on indexed input/output and the differences between formatted and unformatted input/output.

8.16 READ (Internal)

Purpose

Transfers data from an internal file to internal storage.

Syntax

Formatted

```
 READ ([UNIT=]unum,f[,IOSTAT=rn][,ERR=s]
+   [,END= eof]) [iolist]
```

List-Directed

```
 READ ([UNIT=]unum,* [,IOSTAT=rn][,ERR=s]
+   [,END= eof]) [iolist]
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
f	A format specifier
*	List-directed input specifier.
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.
END=*eof*	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 and Input/Output List sections in this chapter for details on these parameters.

See the Data Transfer Rules section of this chapter and Chapter 7 for more information on formatted and list-directed input/output. Chapter 7 also contains an example input/output using internal files.

Note: The DECODE statement can also be used to control internal input. See the DECODE statement description in this chapter for more information.

8.17 READ (Sequential)

Purpose

Transfers data from an external record to the items specified by the input list. Transfers occurs using the sequential access method or keyed access method. (See Chapter 7.)

Syntax

Formatted

```
 READ ([UNIT=]unum,f[,IOSTAT=rn][,ERR=s]
+   [,END= eof]) [iolist]
 READ f[,iolist]
```

List-Directed

```
 READ ([UNIT=]unum,*[,IOSTAT=rn][,ERR=s]
+   [,END= eof])[iolist]
 READ f*[iolist]
```

Unformatted

```
 READ ([UNIT=]unum[,IOSTAT=rn][,ERR=s]
+   [,END=eof]) [iolist]
```

Namelist-Directed

```
  READ (unum,NML=group-name[,IOSTAT=rn][,ERR=s]
 +   [,EN D=eof]) [iolist]
  READ name
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
f	A format specifier.
*	List-directed input specifier.
NML= *group-name*	A namelist specifier
name	Same as above, except the keyword NML is omitted
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.
END=*eof*	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 and Input/Output List sections in this chapter for details on these parameters.

Method of Operation

Formatted

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

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.

Namelist-Directed

A namelist-directed READ statement locates data in a file using the group name in a NAMELIST statement (see Chapter 4). It uses the data types of the items in the corresponding NAMELIST statement and the forms of the data to edit the data. See Namelist-Directed Rules in this section for more information on this format.

Unformatted

A unformatted READ statement transfers data from an external record to internal storage. The READ operation performs no translation on read-in data. The data in 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*.

Rules

Unformatted

1. 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.

2. The type of each data item in the input record must match the corresponding data item in *iolist*.

List-Directed

1. 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 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.

2. Hollerith, octal, and hexadecimal constants are not allowed.

3. 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.

4. A space, tab, comma, or slash appearing within a character constant are processed as part of the constant, and not as delimiters.

5. 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.

6. When the external record specified contains character constants, a slash must be specified to terminate processing of the record. 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.

7. Each READ reads as many records as is required by the specifications in *iolist.* Any items in a record appearing after a slash are ignored.

Namelist-Directed

1. The figure below gives some rules for namelist input data and shows its format:

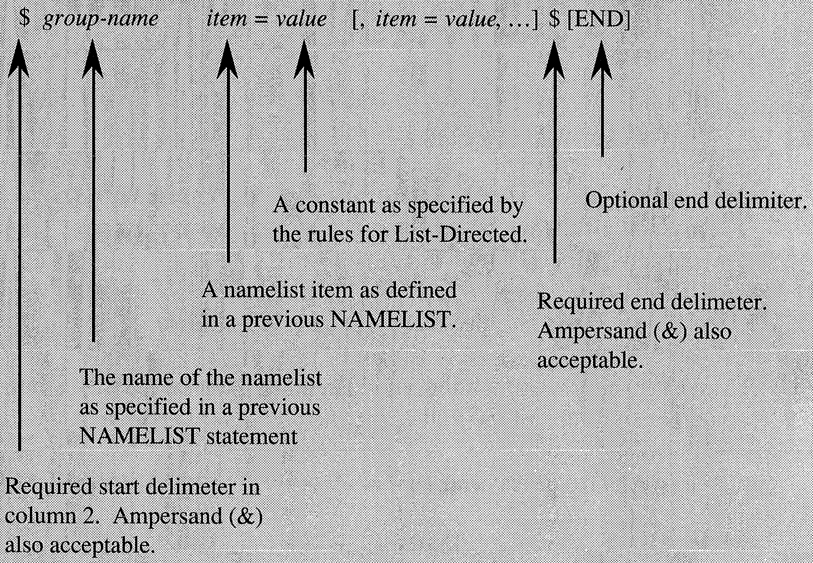

$ *group-name* *item* = *value* [, *item* = *value*, ...] $ [END]
A constant as specified by the rules for List-Directed.
Optional end delimiter.
A namelist item as defined in a previous NAMELIST.
Required end delimeter. Ampersand (&) also acceptable.
The name of the namelist as specified in a previous NAMELIST statement
Required start delimeter in column 2. Ampersand (&) also acceptable.

Figure 8–1. Namelist Input Data Rules

2. Both *group-name* and *item* must be contained within a single record.

3. Spaces and/or tabs aren't allowed within *group-name* or *item*. However, *item* can contain spaces or tabs within the parentheses of a subscript or substring specifier.

4. The *value* item can any of the values given under Rule 1 in the previous section, List-Directed.

5. A *value* separator must delimit each item in a list of constants. See Rules 3 and 4 under List-Directed.

6. A separator must delimit each list of value assignments. See Rule 3 under List-Directed. Any number of spaces or tabs can precede the equal sign.

7. 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.

8. Entering a question mark (?) after execution of a namelist-directed READ statement is executed causes the group-name and current values of the namelist items for that group to be displayed.

9. 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.

10. You can assign input values for the following data types: integer, real, logical, complex, and character. Table 5.1 in Chapter 5 gives the rules for conversion when the data type of the namelist item and the assigned constant value don't match.

11. Numeric-to-character and character-to-numeric conversions are not allowed.

12. Constant values must be given for assigned values, array subscripts, and substring specifiers. Symbolic constants defined by a PARAMETER statement are not allowed.

Examples

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 the Data Transfer Rules section of this chapter and Chapter 7 for more information on formatted, list-directed unformatted, and namelist-directed input/output.

8.18 REWIND

Use

Positions a file at its initial point. It may be used with both unformatted and formatted data files.

Syntax

```
REWIND u
REWIND (alist)
```

where *u* is an external unit identifier and *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 *input/output 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 to be 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 (exception: see *unit specifier*). 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. A REWIND statement for a file that is connected but does not exist is allowed but has no effect.

Examples

```
REWIND 8
REWIND (UNIT=NFILE,ERR=555)
```

8.19 REWRITE

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]
```

Unformatted

```
   REWRITE ([UNIT=]unum[,IOSTAT=rn][,ERR=s]) [iolist]
```

Parameter Explanations

[UNIT=]*unum*	Unit or internal file to be acted upon.
f	A format specifier.
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.

See Control Information List and Input/Output List sections in this chapter for details on these parameters.

Rules of 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 the Data Transfer Rules section of this chapter for additional rules.

Example

```
    REWRITE (10), A,B,C
```

Rewrites the last record accessed to the indexed file connected to logical unit 10.

8.20 UNLOCK

Purpose

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]
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.

See the Control Information List section in this chapter 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.

Examples

```
    UNLOCK (10)
```

unlocks the last record read in from the file connected to logical unit 10.

8.21 WRITE (Direct Access)

Purpose

Transfers data from internal storage to an external indexed using the direct access method.

Syntax

Formatted

```
     WRITE ([UNIT=]unum,REC=rn,f[,IOSTAT=rn]
    +    [,ERR=s]) [iolist]
```

Unformatted

```
     WRITE ([UNIT=]unum,REC=rn,[,IOSTAT=ios]
    +    [,ERR= s]) [iolist]
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
REC=*rn*	Direct access mode. The number of the record to be accessed.
f	A format specifier
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.
iolist	Specifies memory location from which data is to be written.

See Control Information List and Input/Output List sections in this chapter for details on these parameters.

See the Data Transfer Rules section of this chapter and Chapter 7 for more information on formatted and unformatted input/output.

Rules of Use

1. Execution of a WRITE statement for a file that does not exist creates the file.

8.22 WRITE (Indexed)

Purpose

Transfers data from internal storage to external records using the keyed access method.

Syntax

Formatted

```
    WRITE ([UNIT=]unum,f[,IOSTAT=rn][,ERR=s]) [iolist]
```

Unformatted

```
    WRITE ([UNIT=]unum[,IOSTAT=rn][,ERR=s]) [iolist]
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
f	A format specifier.
*	List-directed out specifier.
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.
iolist	Specifies memory location from which data is to be written.

See Control Information List and Input/Output List sections in this chapter for details on these parameters.

See the Data Transfer Rules section of this chapter and Chapter 7 for more information on formatted and unformatted input/output.

Rules of Use

1. Execution of a WRITE statement for a file that does not exist creates the file.

8.23 WRITE (Internal)

Purpose

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]
```

List-Directed

```
    WRITE ([UNIT=]unum, *[,IOSTAT=rn][,ERR=s])[iolist]
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
f	A format specifier.
*	list-directed out specifier.
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.
iolist	Specifies memory location from which data is to be written.

See Control Information List and Input/Output List sections in this chapter for details on these parameters.

See the Data Transfer Rules section of this chapter and Chapter 7 for more information on formatted and list-directed input/output. Chapter 7 also contains an example input/output using internal files.

Rules of Use

1. Execution of a 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 in this chapter for more information.

8.24 WRITE (Sequential)

Purpose

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=rn][,ERR=s]) [iolist]
```

List-Directed

```
    WRITE ([UNIT=]unum,*[,IOSTAT=rn][,ERR=s]) [iolist]
```

Unformatted

```
    WRITE ([UNIT=]unum[,IOSTAT=rn][,ERR=s]) [iolist]
```

Namelist- Directed

```
    WRITE([UNIT=]unum,NML=group-name[,IOSTAT=rn]
+      [,ERR =s][,END=eof])
```

Parameter Explanations

UNIT=*unum*	Unit or internal file to be acted upon.
NML= *group-name*	A namelist specifier.
f	A format specifier.
*	List-directed out specifier.
REC=*rn*	Direct access mode. The number of the record to be accessed.
IOSTAT=*rn*	Name of variable in which I/O completion status is to be posted.
ERR=*s*	Statement label to which control is transferred after an error.
iolist	Specifies memory location from which data is to be written

See Control Information List and Input/Output List sections in this chapter for details on these parameters.

See the Data Transfer Rules section of this chapter and Chapter 7 for more information on formatted, list-directed, and unformatted input/output.

Method of Operation

Formatted

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.

List-Directed

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.

Namelist-Directed

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). A namelist-directed READ or ACCEPT statement can read the output of a namelist-directed WRITE statement.

Unformatted

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.*

Rules

Execution of a WRITE statement for a file that does not exist creates the file.

List-Directed

1. The item to be transferred to an external record can be a constant with a data type of integer, real, logical, complex, or character.

2. The rules given in Chapter 2 define the acceptable formats for constants in the external record, except for character constant. A character constant does not require delimiting apostrophes; an apostrophe within a character string is represented by one, instead of two, apostrophes.

3. The table below shows the data types and the defaults of their output format.

Data Type	Format Specification of Default Output
LOGICAL*1	I5
BYTE	L2
LOGICAL*2	L2
LOGICAL*4	L2
LOGICAL	L2
INTEGER*2	I7
INTEGER*4	I12
INTEGER	I12
REAL*4	1PG15.7E2
REAL	1PG15/7E2
REAL*8	1PG24.16E2
REAL*16	1PG43.33E4
COMPLEX	'(',1PG15.7E2,',',1PG15.7E2,')'
COMPLEX*16	'(',1PG24.16E2,',',1PG24.16E2,')'
CHARACTER**n*	A*n*, where *n* is the length of the character expression

Table 8-7. Default Formats of List-Directed Output

4. List-directed character output data cannot be read as list-directed input because of the different use of apostrophes described in Rule 2.

5. A list-directed output statement can write one or more records. Position 1 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

6. The output of a complex value contains no embedded spaces.

7. 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

Namelist items are written in the order that referenced NAMELIST defines them.

Examples

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)
```

8.25 Control Information List — cilist

This section describes the components of the control information list (*cilist*) and the input/output list (*iolist*), which can be specified as elements of the I/O statements described in this chapter.

The table below summarizes the items that can be specified in a *cilist*. Each cilist specifier shown in the table can appear at most 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:

1. The keyword UNIT= may be omitted if and only if the unit specifier is the first item on the list.

2. The keyword FMT= may 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 input/output operation; default is an unformatted input/output operation. If a record specifier is present, then direct access input/output is denoted; default is sequential access.

Specifier	Purpose
[UNIT=] *u*	Unit or internal file to be acted upon.
[NML= *group-name*]	Identifies the *group-name* of a list of items when namelist-directed I/O is used
[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 to be accessed in direct access mode.
KEY [ ] *c*= *val*	Value of the key field in a record to be 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 to be referenced in indexed access mode.
IOSTAT= *ios*	Name of a variable in which I/O completion status is to be 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.

Table 8-8. Control Information List Specifiers

8.25.1 Unit Specifier — unum

The form of a *unit specifier* is:

```
[UNIT=] u
```

where *u* is a unit identifier specified as follows:

1. A non-negative integer or non-integer expression specifying the unit.

   A non-integer expression is converted to integer, and the fractional portion, if present, is discarded prior to use.

2. 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.

3. 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 Rule 1 or 2 above, but it cannot be an asterisk in an auxiliary output statement as described in Rule 2.

An *internal file identifier* must be specified as described in Rule 3.

The syntax shows that the keyword UNIT= may be omitted. 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)
```

8.25.2 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 in second in a control information list, and the UNIT= keyword must also have been omitted.

The legal kinds of format identifiers are:

1. The statement label of a FORMAT statement (the FORMAT statement and the format identifier must be in the same program unit).

2. 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).

3. A character expression (provided it does not contain the concatenation of a dummy argument that has its length specified by an asterisk).

4. The name of a character array.

5. An asterisk that is used to indicate list-directed formatting.

8.25.3 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 (Chapter 4). NML can be omitted when preceded by unit specifier (*unum*) without the optional UNIT keyword.

8.25.4 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 input/output operation. Record numbers must be integers greater than zero.

8.25.5 Key-Field-Value Specifier — KEY

The indexed access method use the key-field-value specified in a READ, REWRITE, or other input/output statement, and a key field in the record 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 parameter used in the OPEN statement that created the file.

The key-field-value specifier has the forms shown in the following table:

Specifier	Basis for Record Selection
`KEY=` *kval* `KEYEQ=` *kval*	The key-field value *kval* and the key field are equal
`KEYGT=` *kval*	The key-field value is greater than the key field
`KEYGE=` *kval*	The key-field value is greater than or equal to the key field

Table 8-9. Forms of the Key-Field-Value Specifier

The following rules apply to *kval*:

1. *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.

2. The character expression can be an ordinary character string or an array name of type LOGICAL*1 or BYTE containing Hollerith data.

3. The character or integer type specified for *kval* must match the type specified for the key field in the record.

8.25.6 Key-of-Reference Specifier — KEYID

The key-of-reference specifier designates in a READ, REWRITE, or other input/output 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, etc. The KEY parameter of the OPEN statement that created the files creates and establishes the ordering of the primary and alternate keys.

If KEYID isn't specified, the previous KEYID specification in an input/output statement to the same input/output unit is used. The default for KEYID is zero (0) if it isn't specified for the first input/output statement.

8.25.7 Input/Output Status Specifier — *ios*

An *input/output 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 input/output statement containing this specifier causes *ios* to become defined with the value:

1. Zero if neither an error condition nor an end-of-file condition is encountered by the processor, indicating a successful operation.

2. Positive integer if an error condition occurred.

3. Negative integer if an end-of-file condition is encountered without an error condition.

8.25.8 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 input/output statement with an error specifier, execution of the statement is terminated and the file position becomes indeterminate. If the statement contains an input/output input/output status specifier, the status variable *ios* becomes defined with a processor dependent positive integer. Execution then continues at the statement labeled *s*.

8.25.9 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 input/output status specifier, the input/output status variable *ios* becomes defined with a processor-dependent negative integer. Execution then continues at the statement labeled *s* .

8.26 Input/Output List — *iolist*

This section describes the components of input/output 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 input/output statements READ, WRITE, and PRINT.

If an array name is given as an input/output list item, the elements in the array are treated as though each element were explicitly specified in the input/output list in storage order. Note that the name of an assumed-size dummy array (i.e. an array declared with * for an upper bound) must not appear as an input/output list item.

8.26.1 Input List

An input list item can be one of the following:

- A variable name
- An array element name
- A substring name
- An array name
- 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 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
```

8.26.2 Output List

An output list item can be one of the following:

- A variable name
- An array element name
- A substring name
- An 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
```

8.26.3 Implied-DO Lists

An implied-DO list in a specification that follows the input/output list (*iolist*) in an input/output 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 upon; *i* is an iteration count; and *e1, e2,* and *e3* are control parameters. See the description of the DO statement in Chapter 6 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*.

Example

The following statements write *Hello World* to the standard output 100 times:

```
          write (*,111) ('Hello World',i=1,100)
    111   format (1x,A)
          end
```

8.27 Data Transfer Rules

Data are transferred between records and items specified by the input/output list. The list items are processed in the order in which they appear in the list.

The following restrictions apply to data transfer operations:

1. An input list item must not contain any portion of the established format specification.

2. If an internal file has been specified, an input/output list item must not be in the file or associated with the file.

3. Each output list item must be defined prior to the transfer of that item.

4. All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

The following sections discuss the rules specific to unformatted and formatted input/output.

8.27.1 Unformatted Input/Output

The execution of an *unformatted* input/output statement transfers data without editing between the current record and the items specified in the input/output 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:

1. 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 undefined.

2. 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 input/output.

8.27.2 Formatted Input/Output

The execution of a *formatted* input/output statement transfers data with editing between the items specified by the input/output 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 is 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 input/output 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. Formatted input/output is explained in more detail in Chapter 9.

9. Format Specification

9.1 Overview

A format specification provides explicit editing information to the processor on the structure of a formatted data record. It is used with formatted input/output statements to allow conversion and data editing under program control. An asterisk (*) used as a format identifier in an input/output 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 Chapter 4 of the *FORTRAN 77 Programmer's Guide.*

Format specifications can be given in two ways: in FORMAT statements and as values of character arrays, character variables, and other character expressions.

9.2 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, upon 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 via a character array name may fill the whole array, while those read via a character array element name must fit in a single element of that array.

9.3 FORMAT Statement

The FORMAT statement is a nonexecutable 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 the next section).

9.3.1 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.
ffd	is a *repeatable edit descriptor*, or a *field descriptor.*
efd	is a *non-repeatable edit descriptor.*
fs	is a *format group* and has the same form as a complete *format specification* (described above) 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* may be omitted as follows:

- Between a P edit descriptor and immediately following an F, E, D, or G edit descriptor (see the section on the P Edit Descriptor later in this chapter).
- Before or after a slash edit descriptor (seethe section on Slash Editing later in this chapter).
- Before or after a colon edit descriptor (see the section on the Colon Descriptor later in this chapter).

9.3.2 Repeatable Descriptors

Some *descriptors* may be repeated, others may not.

The repeatable descriptors are:

```
Iw[.m]
Ow[.m]
Zw[.m]
Fw.d
Ew.d[Ee]
Dw.d
Gw.d[Ee]
Lw
A[w]
Q
```

where *w* and *e* are nonzero, unsigned integer constants; and *d* and *m* are unsigned integer constants. Their meanings are given in the section that describes the individual descriptors.

The non-repeatable descriptors are:

```
/  kP   TRc  SS  nHh...   $
:  Tc   S    BN  'h... '
nX TLc  SP   BZ  "h... "
```

where *n* and *c* are nonzero, unsigned integer constants; *k* is an optionally signed integer constant; and *h* is one of the characters capable of representation by the processor.

9.3.3 Format Specifier Usage

Each *field descriptor* corresponds to a particular data type input/output list item:

- Integer field descriptors - Iw, Iw.m, Ow, Zw
- Real, Double Precision, and Complex field descriptors - Fw.d, Ew.d, Ew.dEe, Dw.d, Gw. d, Gw.dEe
- Logical field descriptor - Lw
- Character and Hollerith field descriptors - A, Aw

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 non-zero. *k* is an optionally signed integer constant. Their meanings are given in the section that describes the individual field descriptors.

r, the repeat specifier, 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 9.1 is an alphabetical summary and brief explanation of the 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 zeroes
D*w*.*d*	Transfers real values (D exponent field indicator)
E*w*.*d*[Ee]	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
*k*P	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	Supresses plus characters (+) for positive values in numeric output fields
T*c*	Specifies positional tabulation
TL*c*	Specifies relative tabulation (left)
TR*c*	Specifies relative tabulation (right)
*n*X	Specifies that *n* column positions are to be skipped
Z*w*[.*m*]	Transfers hexadecimal integer values

Table 9-1. Summary of Field and Edit Descriptors

Form	Effect
:	Terminates format control if the I/O list is exhausted
/	Record terminator
$	Specifies suppression of line terminator on output (ignored on input)

Table 9–1 (continued). Summary of Field and Edit Descriptors

Use of each of these field descriptors is described in the following sections.

9.3.4 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 FORMAT statement. Any expression may be enclosed in angle brackets (<>), and used in the same way that an integer constant would be 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 12 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 integer 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 input/output operation.

9.3.5 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.

1. A FORMAT statement must always be labeled.

2. 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; they cannot be symbolic names of constants. The repeat count *r* can be omitted.

3. 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* mu st be specified. In a field descriptor such as E*w*.*d*E*e* , the term *e* must also be an unsigned non-zero integer constant.

4. In an H edit descriptor such as *n*Hc1 c2.\ .\ .c sub *n* , exactly *n* characters must follow the H. Any character in the processor character set can be used in this edit descriptor.

5. 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.

6. 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*.

7. 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.

8. 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.

9. The first character in an output record generally contains carriage control information. See the sections titled Output Rules Summary and Carriage Control later in this chapter.

10. A slash (/) is both a format specifier list separator and a record terminator. See the section on slash editing later in this chapter.

11. 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.

9.3.6 Input Rules Summary

There are certain guidelines that are applicable specifically on input:

1. A minus sign (-) must precede a negative value in an external field; a plus sign (+) is optional before a positive value.

2. 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 later in this chapter).

3. 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 later in this chapter).

4. 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.

5. If an external field contains an exponent, the current scale factor *k*P descriptor has no effect for the conversion of that field.

6. The format specification together with the input list must not attempt to read beyond the end of a record.

9.3.7 Output Rules Summary

There are certain guidelines that are applicable specifically on output:

1. A format specification cannot specify more output characters than the value in the record length specifier (see OPEN). For example, a line printer record might be limited to no more than 133 characters, including the carriage control character.

2. 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.

3. 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** later in this chapter).

9.4 Field and Edit Descriptors

The *format specifiers* in a format specification consist of *field*, or repeatable descriptors, and other, *non-repeatable 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, *non-repeatable 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).

The following sections describe each of the edit descriptors in detail.

9.5 Field Descriptor Reference

9.5.1 Numeric Field Descriptors

The I, O, Z, Q, F, E, D, and G field descriptors described here are used for numeric editing. The P scale factor, also described, can alter the effect of F, E, D, and G *field descriptors.*

Unless otherwise indicated, the following rules apply:

1. 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.

2. 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.

3. 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.

4. 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.

9.5.2 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 don't specify a value, the default values shown in the following table apply. The length of the I/O variable determines the length *n* for the A field descriptor.

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*		

Table 9–2. Default Field Descriptors

9.5.3 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:

```
Iw[.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 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.

Input Example:

Format	External Field	Internal Value
I4	3244	3244
I3	-15	-15
I9	213	213

Table 9–3. I Field Input Examples

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 record right-justified 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 leftmost, 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.

Output Example:

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`	

Table 9–4. I Field Output Examples

9.5.4 O Field Descriptor

The O field descriptor transfers data values and converts them to octal form. It has the form:

```
    Ow[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 the following format:

- 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.

Input Example:

BN is assumed in effect, and internal values are expressed in decimal (base 10).

Format	External Field (INTEGER*4)	Internal Value
`O20`	`-77`	`-63`
`O20`	`1234`	`668`
`O20`	`177777`	`65535`
`O20`	`100000`	`32768`

Table 9–5. O Field Input Examples

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.

Output Example:

Format (INTEGER*4)	Internal Value	External Field
`O20.2`	`3`	`03`
`O20.2`	`-1`	`37777777777`
`O3`	`-1`	`***`
`O20.2`	`63`	`77`
`O20.2`	`-2`	`37777777776`

Table 9–6. O Field Output Examples

9.5.5 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 the following form:

- 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 treated as the value zero.

Input Example:

BN is assumed in effect, and internal values are expressed in decimal (base 10).

Format	External Field (INTEGER*4)	Internal Value
Z10	-FF	-255
Z10	1234	4660
Z10	FFFF	65535
Z10	8000	32768

Table 9–7. Z Field Input Examples

Output Example:

Format (INTEGER*4)	Internal Value	External Field
Z10.2	3	" 03"
Z10.2	-1	" FFFFFFFF"
Z10.2	63	" 3F"
Z10.2	-2	" FFFFFFFE"

Table 9–8. Z Field Output Examples

9.5.6 F Field Descriptor

The F field descriptor transfers real values. It has the form:

```
Fw.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, DOUBLE PRECISION, or COMPLEX data type.

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, and assigns the real value to the corresponding I/O list element. Refer to Real Types in Chapter 2 for more information. If the external field contains an exponent, the letter E may 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 treated as 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 rightmost *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 *kP* (see Scale Factor later in this chapter) 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.

Input Example:

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`

Table 9–9. F Field Input Examples

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.

Output Example:

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`

Table 9–10. F Field Output Examples

9.5.7 E Field Descriptor

The E field descriptor transfers real values in exponential form. It has the form:

```
Ew.d[Ee]
```

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, DOUBLE PRECISION, 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.

Input Example:

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`

Table 9–11. E Field Input Examples

Note that in the last example, 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 record right-justified 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:

- A 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
- A 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 in the range of -99 to +99
E*w.d*	+*nnn* or -*nnn* if the value of the exponent is <= -99 or <= +99
Ew.dEe	E + n1 n2... n sub e or E - n1 n2... n sub e, where n1 n2... n<F8>e 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 (see later in this chapter), w is $\geq d + 7$, or $\geq d + e + 5$ if e is present.

Output Example:

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"`

Table 9–12. E Field Output Examples

9.5.8 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, DOUBLE PRECISION, 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.

Input Example:

Format	External Field	Internal Value
D10.2	"12345 "	.12345E+03
D10.2	" 123.45"	.12345E+03
D15.3	"123.4567891D+04"	.1234567891E+07

Table 9–13. D Field Input Examples

In an output statement, the D field descriptor has the same effect as the E field descriptor, except that the D exponent field indicator is used in place of the E indicator.

Output Example:

Format	Internal Value	External Field
D14.3	123D-04	" .123D-04"
D23.12	123456789123D+04	" .123456789123D+04"
D9.6	14D+01	"*********"

Table 9–14. D Field Output Examples

9.5.9 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 beforehand. On output, the G field descriptor produces a field as with 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:

```
    Gw.d[Ee]
```

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, DOUBLE PRECISION, 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 record right-justified 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.

Data Magnitude	Effective Format
$m < 0.1$	`Ew.d[Ee]`
$0.1 \leq m < 1.0$	`F(w-n).d, n ('')`
$1.0 \leq m < 10.0$	`F(w-n).(d-1) ('')`
•	•
•	•
•	•
$10^{d-2} \leq m < 10^{d-1}$	`F(w-n).1,n ('')`
$10^{d-1} \leq m < 10^{d}$	`F(w-n).0n ('')`
$m \geq 10^{d}$	`Ew.d[Ee]`

Table 9–15. Effect of Data Magnitude on G Format Conventions

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 ≥ $d + 7$, or ≥ $d + e + 5$.

Output Example:

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"`

Table 9–16. G Field Output Examples

The following examples use the same values with an equivalent F field descriptor, for comparison purposes:

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	"*************"

Table 9–17. Field Comparison Examples

9.5.10 P Edit Descriptor

The P edit descriptor specifies a *scale factor* and has the form:

```
kP
```

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:

```
kPFw.d      kPEw.d kPD w.dkPGw.d
```

The value of *k* must not be greater than *d*+1, where *d* is a 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 10^k
- 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 10^k 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 2P 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.

Format	External Field	Internal Value
`3PE10.5`	`"    37.614"`	`.37614E-01`
`3PE10.5`	`"  37.614E2"`	`.37614E+04`
`-3PE10.5`	`"    37.614"`	`.37614E+05`

Table 9–18. Scale Factor Examples

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 10^k 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 10^k 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 10^k, 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

Output Example:

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"

Table 9–19. Scale Format Output Examples

On output, the effect of the scale factor for the G field descriptor is suspended if the magnitude of the data to be output 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 for 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 10. 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 initially 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 0P specification.

9.5.11 L Edit Descriptor

The L edit descriptor is used for logical data. The specified input/output list item must be of type LOGICAL. It has the form:

```
Lw
```

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 may 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.

Format	Internal Value	External Field
`L5`	`.TRUE.`	`"    T"`
`L1`	`.FALSE.`	`"F"`

Table 9–20. L Field Examples

The L edit descriptor can also be used to process integer data items. All non-zero values are displayed as .TRUE. and all zero values as .FALSE..

9.5.12 A Edit Descriptor

The A edit descriptor is used for editing of character or Hollerith data.

```
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 of any data type. If it is of character data type, character data is transmitted. If it is of 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 follows:

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

Table 9–21. I/O List Element Sizes

If *w* is greater than the maximum number of characters that can be stored in the corresponding I/O list item, only the rightmost characters of the field are assigned to that element. The leftmost 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, left-justified, and trailing blanks are added to fill it to its maximum size.

Input Example:

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)

Table 9–22. A Field Input Examples

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 leftmost *w* characters from the I/O list item are transferred.

Output Example:

Format	Internal Value	External Field
A6	"GREEK"	" GREEK"
A6	"FRENCH"	"FRENCH"
A6	"PORTUGUESE"	"PORTUG"

Table 9–23. A Field Output Examples

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 of character type, the default value is the length of the I/O list element. If it is of 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.

9.5.13 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, a specification of the form 4F5.2 is equivalent to F5.2, F5.2, F5.2, F5.2.

Enclosing a group of field descriptors in parentheses, and preceding the enclosed group with a repeat count, repeats the entire group. Thus, 2(I6,F8.4) is equivalent to I6,F8.4,I6,F8.4.

9.5.14 H Field Descriptor

The H field descriptor is used for output of character literal data. It has the form:

```
nHxxx... 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.

Output Example:

Specification	External Field
`6HAb CdE`	`Ab CdE`
`1H9`	`9`
`4H'a2'`	`'a2'`

Table 9–24. H Edit Description Output Examples

An H field descriptor must not be encountered by a READ statement.

9.5.15 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:

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"`

Table 9–25. Character Edit Description Examples

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.

9.5.16 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 of 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:

```
        INTEGER N
        CHARACTER LINE * 80
        READ (5, 100) N, LINE (1:N)
  100   FORMAT (Q, A)
```

9.6 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.

9.6.1 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:

```
nX
```

where *n* is an unsigned, nonzero integer constant denoting the number of characters to be skipped.

9.6.2 T Edit Descriptor

The T edit descriptor specifies an absolute position in an input or output record. It has the form:

```
Tn
```

where *n* indicates the next character transferred to or from the record is the *n*th character of the record.

9.6.3 TL Edit Descriptor

The TL edit descriptor specifies a position to the left of the current position. It has the form:

```
TLn
```

where *n* indicates that the next character to be transferred from or to the record is the nth 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.

9.6.4 TR Edit Descriptor

The TR edit descriptor specifies a position to the right of the current position. It has the form:

```
TRn
```

where *n* indicates that the next character to be transferred from or to a record is a *n*th character to the right of the current character. The value of *n* must be greater than or equal to one.

9.6.5 BN Edit Descriptor

The BN edit descriptor causes the processor to ignore blank characters in a numeric input field and to right-justify 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.

9.6.7 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.

9.6.8 SP Edit Descriptor

The SP edit descriptor specifies that a plus sign should be inserted in any character position that normally contains an optional plus sign and whose actual value is ≥ 0. It has the form:

```
    SP
```

The SP descriptor affects only I, F, E, D, and G editing, and then only on output fields.

9.6.9 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.

9.6.10 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 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.

9.6.11 Colon Descriptor

The colon character (:) in a format specification terminates format control if no more items are in the I/O list. The : descriptor has no effect if I/O list items remain.

9.6.12 $ Edit Descriptor

The $ edit descriptor suppresses the terminal linemark character at the end of the current output record. It has the form:

```
$
```

The $ descriptor is non-repeatable and is ignored when encountered during input operations.

Output Example:

```
      PRINT 100, 'ENTER A NUMBER:'
100   FORMAT (1X, A, $)
      READ *, X
```

9.7 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 input/output 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 input/output 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.

Input Example:

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)`

Table 9–26. Complex Data Editing Input Examples

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.

Output Example:

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, *****"`

Table 9–27. Complex Data Editing Output Examples

9.7.1 Carriage Control

A formatted record can contain a prescribed carriage control character as the first character of the record. The carriage control character is the first character of the record and determines vertical spacing in printing when the CARRIAGE-CONTROL keyword of the OPEN statement is set to FORTRAN (the default). The carriage control characters are:

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. Doesn't return to the left margin after printing

Table 9–28. Carriage Control Characters

The carriage control character is not printed and the remaining characters, if any, are printed in 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.

9.7.2 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 causes *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.

9.8 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, and
- The next item in the input/output list, if one exists.

If an input/output 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 ( ) may 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*, and a format specification preceded by a repeat specification, *r* (*flist*), a format specification is interpreted from left to right (see the section on 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 input/output 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 input/output list, and format control communicates information directly with the record.

Whenever format control encounters a *repeatable edit descriptor* in a format specification, it determines whether there is a next item in the input/output 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 next item, format control terminates.

If format control encounters the rightmost 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 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 the Scale Factor description in the **P Edit** section of this chapter), the S, SP, or SS edit descriptor sign control, or the BN or BZ edit descriptor blank control.

9.9 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. The rules for list-directed external input/output and internal input/output both apply to internal list-directed input/output.

9.9.1 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, a null value (see Note 5 below), 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 and *c* is a constant.

The *r** form is equivalent to *r* successive null values. Neither form may contain embedded blanks, except where permitted within the constant *c*.

Data values may be separated by 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:

1. 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.

2. 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.

3. For data of type logical, the input form must not include either slashes or commas among the optional characters allowed for L-editing.

4. For data of type character, the input form is a character constant: a nonempty 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 may be continued on as many records as needed. Constants are assigned to list items as in character assignment statements.

5 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 may not be used as either the real or imaginary part alone.

6. 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 given in the input statement. For example,

```
(i10, f20.10,i4)
```

reads the following record correctly:

```
-345,.05e-3,12
```

9.9.2 List-Directed Output

The form of the values produced is the same as that required for input, except as noted below:

1. Logical output constants are T for the value true and F for the value false.

2. 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.

3. 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.

4. Character constants produced 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.

5. Slashes and null values are not produced but each record begins with a blank character to provide carriage control if the record is printed.

6. Two non-character values in succession in the same record will be separated by a value separator consisting of one or more blanks. No value separator is produced before or after a character value.

10. Functions and Subprograms

This chapter is divided into the following major sections:

- Statement Functions, which gives the syntax and rules for defining statement functions.
- Parameter Passing, which gives general rules in passing parameters to functions and subprograms.
- Function and Subroutine Programs, which gives rules for writing and using function and subroutine subprograms.
- The syntax and rules for the FUNCTION, SUBROUTINE, ENTRY, and INCLUDE statements, used to specify function and subroutine subprograms.

10.1 Overview

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 FORTRAN compiler. This chapter discusses user-written subprograms; compiler-supplied functions and subroutines are discussed in Appendix A.

There are three types of program units:

- *statement functions*, which consist of a single arithmetic statement defined within the main program unit or a subprogram.
- *function subprograms*, which consist of one or more statements defined external to the main program unit. It is invoked when referenced as a primary in an expression contained in another program unit.
- *subroutine subprograms*, which consist of one or more program statements defined external to the main program unit. It is invoked when referenced in a CALL (Chapter 6) statement in another program unit.

10.2 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.

10.2.1 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* may include:

- Constants
- Symbolic names of constants
- Variable references
- Array element references
- Library function references
- Reference 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 statement function statement 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 statement function statement with the subprogram.

A dummy argument that appears in an ENTRY statement may be referenced in the expression of the statement function statement only if the dummy argument name appears in a FUNCTION, SUBROUTINE, or ENTRY statement preceding the statement function definition.

10.2.2 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.

10.2.3 Operational Conventions and Restrictions

The expressions must agree in order, number, and type with the corresponding dummy arguments. The expressions may 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 may be referenced only in the program unit that contains its definition. A statement function can reference another statement function that has been defined prior to the referencing function, but not one which 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 must not appear as a symbolic name in any specification statement except in 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 statement.

- The symbolic name of a statement function can not be an actual argument, and must not appear in an EXTERNAL statement.

- A statement function statement in a function subprogram must not 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.

10.3 Parameter Passing

10.3.1 Dummy 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 name in FUNCTION, SUBROUTINE, ENTRY, or statement function statements in the same program unit.

Actual arguments are the items which are given 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 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 may also have adjustable dimensions.

10.3.2 Built-In Functions

Built-in functions permit you to communicate with non-FORTRAN programs that require arguments passed in a specific format. (Seep See Chapter 3 of the *FORTRAN 77 Programmer's Guide* for information specific to communicating with programs written in the C and Pascal languages.)

The built-in functions %VAL, %REF, and %DESCR are intended for use with arguments within an argument list. The built-in function %LOC is intended for global use.

%VAL

The %VAL built-in function passes the argument as a 32-bit value; the function extends argument 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 %REF built-in function passes an argument by reference; it has the syntax:

```
%VAL(a)
```

where *a* is an argument within an argument list.

%DESCR

The %DESCR built-in 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 %LOC built-in 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.

10.4 Function and Subroutine Subprograms

A function subprogram consists of a FUNCTION statement followed by a program body that terminates with an END statement and has the following characteristics:

- it is defined external to the main program unit
- it is referenced as a primary in an expression contained in another program unit
- it is 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 written in other languages.)

A subroutine subprogram consists of a SUBROUTINE statement (described in this chapter), followed by a program body that terminates with an END statement (Chapter 6), and is defined external to the main program.

10.4.1 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 (Chapter 6) 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 via a chain of other subprograms if the automatic storage of variables is in effect. The **–automatic** command line option (Chapter 11) 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 name or a 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 the name must be one of those listed in Appendix A 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.

10.4.2 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, it is this value that 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 may 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, the subroutine and the function, as well, can return values to the calling program unit by defining their dummy arguments during execution.

10.5 FUNCTION

10.5.1 Use

First statement of a function subprogram; specifies the symbolic name of the function and its type.

10.5.2 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 an unsigned, nonzero constant. This specification isn't permitted when the function is type CHARACTER with an explicit length following the keyword CHARACTER.

w is a dummy argument and may be a variable name, array name, or dummy subprogram name.

10.5.3 Rules of Use

1. A FUNCTION statement must appear only as the first statement of a function subprogram.

2. 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.

3. 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.

4. If the function type is specified in the FUNCTION statement, the function name must not appear in a type statement.

5. In the type specification CHARACTER, *len* may 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 of type character. If the length is declared as an asterisk, all such entries must have a length declared with an asterisk.

6. A function specified as a subprogram may be referenced within any other subprogram or the main program of the executable program.

10.5.4 Restrictions

1. A function subprogram must not contain a BLOCK DATA, SUBROUTINE, or PROGRAM statement.

2. A function name must not have its type explicitly specified more than once in a program unit.

3. In a function subprogram, a dummy argument name must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name.

4. 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.

5. The compiler system permits recursion if the automatic storage of variables is in effect. The **–automatic** command line option (Chapter 11) by default causes the automatic storage of variables.

10.6 SUBROUTINE

10.6.1 Use

First statement of a subroutine subprogram.

10.6.2 Syntax

```
SUBROUTINE sub[( [d [,d]...] )]
```

where *sub* is a symbolic name of the subroutine program unit and *d* is a dummy argument and may be a variable name, array name, dummy subprogram name, or an asterisk. The asterisk denotes an alternate return.

10.6.3 Rules of Use

1. A SUBROUTINE statement must be the first statement of a subroutine subprogram.

2. If there are no dummy arguments, either of the following forms may be used:

```
SUBROUTINE sub
SUBROUTINE sub()
```

3. One or more dummy arguments may become defined or redefined to return results.

4. The symbolic name of a subroutine is global and must not be the same as any other global or local name in the program unit.

5. A CALL statement within the body of a subroutine may reference the subroutine itself (recursion) if the automatic storage attribute is specified. See the description of the AUTOMATIC and STATIC keywords in Chapter 4 for more information.

10.6.4 Restrictions

1. A subroutine subprogram must not contain a BLOCK DATA, FUNCTION, or PROGRAM statement.

2. In a subroutine, a dummy argument name is local to the program unit and must not appear in an EQUIVALENCE, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name.

3. A character dummy argument whose length is specified as an asterisk must not appear as an operand for concatenation, except in a character assignment statement.

10.7 ENTRY

10.7.1 Use

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.

10.7.2 Syntax

```
    ENTRY en[( [d [,d]...] )]
```

where *en* is a symbolic name of the entry point and *d* is a dummy argument. If there are no dummy arguments, the following forms may be used:

```
    ENTRY en
    ENTRY en()
```

10.7.3 Method of Operation

Each ENTRY statement in a function or subroutine provides an additional name which you may 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 which provided that name.

Within a function, each of its names (the one provided by the FUNCTION statement, plus the additional 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; but otherwise, the variables need not all have the same data type. Such variables are in effect equivalenced, which has the following implications:

- First, you need not assign the return value to the same name which you used to invoke the function; instead, you may assign to any of the names which has the same data type as that one.

- Second, if you assign to any of the names which does not have the same data type as the one you used to invoke the function, then the return value becomes undefined.

10.7.4 Rules of Use

1. The ENTRY statement may appear anywhere within a function subprogram after the FUNCTION statement or within a subroutine after a SUBROUTINE statement.

2. A subprogram may have one or more ENTRY statements.

3. The entry name *en* in a function subprogram may appear in a type statement.

4. In a function, a local variable with the same name as one of the entries may be referenced.

A subprogram can call itself directly if the automatic storage of variables is in effect. The **–automatic** command line option (Chapter 11) by default causes the automatic storage of variables.

5. The order, number, type, and names of the dummy arguments in an ENTRY statement may 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.

10.7.5 Restrictions

1. 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.

2. 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.

3. 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.

4. 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 (*).

5. 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, appears in a FUNCTION or SUBROUTINE statement, or appears in an ENTRY statement preceding the statement function.

10.8 INCLUDE

Use

Specifies that the contents of a designated file are to be incorporated into the FORTRAN compilation directly following the INCLUDE statement.

Syntax

```
    INCLUDE "filename"
```

where *filename* is a character string constant that specifies the file to be included.

Rules of Use

1. An INCLUDE statement can appear anywhere within a program unit.

2. Upon 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 in the current file with the statement following the INCLUDE statement.

Search Path

Upon encountering an INCLUDE statement, the compiler first searches for a file named *filename*, then for a file named */usr/include/filename*.

Restrictions

1. An included file or module cannot begin with a continuation line. Each FORTRAN statement must be completely contained within a single file.

2. An INCLUDE statement cannon contain continuation lines. The first non-comment line following the INCLUDE statement cannot be a continuation line.

3. An INCLUDE statement cannot be labelled. It must not have a statement number in the statement number field.

11. Compiler Options

This chapter describes options that affect source programs both during compilation and at run-time. Three mechanisms are provided by which to execute these options:

- The $OPTIONS statement, which is specified in the source code as the first statement of a program unit.
- *In-line options*, which are individual statements embedded in the source code.
- The $INCLUDE statement, which allows the inclusion of FORTRAN source statements from an external library to be included 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.*

11.1 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 like-named command line options. See Chapter 1 of the *FORTRAN 77 Programmer's Guide* for a description of these options. An *option* specification overrides a command line option when they are the same; *option* must always be preceded by a slash (/).

The following rules apply to OPTIONS statement specifications:

- the statement must be the first statement in a program unit and must precede the PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA statements.
- *option* remains in effect only for the duration of the program unit in which it is defined.

11.2 In-Line Options

The syntax for in-line compiler options consists of a dollar sign ($) in column one of a source record, followed by the name of the compiler option in either upper or lower case, 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.

Note: The compiler doesn't support the following options, but, for compatibility with other compilers, it does recognize them:

`ARGCHECK`	`NOTBINARY`
`BINARY`	`SEGMENT`
`CHAREQU`	`SYSTEM`
`NOARGCHECK`	`XREF`

Upon encountering one of these options, the compiler issues a warning message and treats it as a comment line.

11.3 $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 Initial Lines in Chapter 1. The compiler command line option **–col72** has an identical effect on a global basis.

11.4 $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.

11.5 $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 an identical effect on a global basis.

11.6 $INCLUDE Option

The **$INCLUDE** option permits the inclusion of source lines from a secondary file (or files) into 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 option 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 option will follow the $INCLUDE option, beginning on the next line. Nesting of $INCLUDE options is permitted within the constraints of the operating system.

11.7 $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 which are assigned a data type either by implicit typing rules, or by use of the 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.

11.8 $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 which are assigned a data type either by implicit typing rules, or by use of the LOGICAL declaration statement without a type length being specified.

Appendix A: Intrinsic Functions

This appendix describe 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.

A.1 Generic and Specific Names

A *generic name* is the single 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 Part 5 of Table A.1). However, the generic name SQRT may be used regardless of the data type of the argument(s).

An intrinsic function preceeded by the letters CD is equivalent to the generic function with the same base name, except that the arguments must of of type DOUBLE COMPLEX.

Intrinsic functions starting with the letter Q are the quad-precision (i.e., REAL*16) versions of the same generic routines. Note that all calculations are still performed in double precision, even when quad-precision storage is used as a place holder.

Intrinsic functions starting with II are equivalent to generic functions with the same base name, except that arguments must of 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 to a *specific name*, depending upon 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 a 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.

A.2 Referencing an Intrinsic Function

Reference to an intrinsic function takes the following 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 may 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 may be used as a primary in an expression. The following example involves referencing an intrinsic function:

```
    X = SQRT(B**2-4*A*C)
```

Arguments for which the result is not mathematically defined or exceeds the numeric range of the processor cause the result of the function to become undefined.

A.3 Operational Conventions and Restrictions

For most intrinsic functions, the data type of the result of the intrinsic function is the same as the argument(s). 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 used 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.

A.4 List of Functions

The tables starting on the next page list the available intrinsic functions. Operational conventions and restrictions (other than those already given) are listed at the end of each 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.

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to integer	1	INT	INT	Real	Integer
				Integer	Integer
			IIFIX	Real*4	Integer*2
			JIFIX	Real*4	Integer*4
			IFIX	Real	Integer
			IDINT	Double	Integer
				Complex	Integer
Conversion to Real	1	REAL	REAL	Integer	Real
			FLOATI	Integer*2	Real*4
			FLOATJ	Integer*4	Real*4
			FLOAT	Integer	Real
			SNGL	Double	Real
				Complex	Real
				Real	Real
Conversion to Double	1	DBLE	DBLE	Integer	Double
			DFLOAT	Real	Double
				Double	Double
				Complex	Double
			DFLOTI	Integer*2	Real*8
			DFLOTJ	Integer*4	Real*8
Conversion to Complex	1, 2	CMPLX	CMPLX	Integer	Complex
				Real	Complex
				Double	Complex
				Complex	Complex
				Complex*16	Complex
Conversion to Complex*16	1, 2	DCMPLX	DCMPLX		

1. INT and IFIX return result type INTEGER*2 if the —i2 compile option is in effect; otherwise, the result type is INTEGER*4.

Table A-1. Intrinsic Functions

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Maximum Value	2 or more	MAX	MAX0	Integer	Integer
		MAX0	IMAX0	Integer*2	Integer*2
			JMAX0	Integer*4	Integer*4
			AMAX0	Integer	Real
		MAX1	MAX1	Real	Integer
			AMAX1	Real	Real
			DAMX1	Double	Double
			IMAX1	Real*4	Integer*2
			JMAX1	Real*4	Integer*4
		AMAX0	AIMAX0	Integer*2	Real*4
			AJMAX0	Integer*4	Real*4
Minimum Value	2 or more	MIN	MIN0	Integer	Integer
		MIN0	IMIN0	Integer*2	Integer*2
			JMIN0	Integer*4	Integer*4
			AMIN0	Integer	Real
		MIN1	MIN1	Real	Integer
			AMIN1	Real	Real
			DAMX1	Double	Double
			IMIN1	Real*4	Integer*2
			JMIN1	Real*4	Integer*4
		AMIN0	AIMIN0	Integer*2	Real*4
			AJMIN0	Integer*4	Real*4

Table A-1 (continued). Intrinsic Functions

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Truncation	1	INT	AINT	Real	Real
			DINT	Double	Double
			IINT	Real*4	Integer*2
			JINT	Real*4	Integer*4
			IIDINT	Real*8	Integer*2
			JIDINT	Real*8	Integer*4
Nearest Whole Number	1	NINT	ININT	Real*4	Integer*2
			JNINT	Real*4	Integer*4
			NINT	Real	Integer
			IDNINT	Double	Integer
		IDNINT	IIDNNT	Real*8	Integer*2
			JIDNNT	Real*8	Integer*4
		ANINT	ANINT	Real	Real
			DNINT	Double	Double
Zero-Extend Functions	1	ZEXT	IZEXT	Logical*1	Integer*2
				Logical*2	
				Integer*2	
			JZEXT	Logical*1	Integer*4
				Logical*2	
				Logical*4	
				Integer*2	
				Integer*4	

1. When NINT or IDNINT is specified as an *argument* in a subroutine call or function reference, the compiler supplies either an INTEGER*2 or INTEGER*4 function depending on the **—i2** command line option (see **Chapter 1** of the *FORTRAN Programmer's Guide*).

Table A-1 (continued). Intrinsic Functions

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Absolute Value	1	ABS	IABS ABS DABS CABS ZABS IIABS JIABS	Integer Real Double Complex DblComplex Integer*2 Integer*4	Integer Real Double Real Double Integer*2 Integer*4
Remaindering	2	MOD	MOD IMOD JMOD AMOD DMOD	Integer Integer*2 Integer*4 Real Double	Integer Integer*2 Integer*4 Real Double
Transfer of Sign	2	SIGN	ISIGN IISIGN JISIGN SIGN DSIGN	Integer Integer*2 Integer*4 Real Double	Integer Integer*2 Integer*4 Real Double
Positive Difference	2	DIM	IDIM IIDIM JIDIM DIM DDIM	Integer Integer*2 Integer*4 Real Double	Integer Integer*2 Integer*4 Real Double
Double Product	2	DPROD	DPROD	Real	Double
Length of Character Entry	1		LEN	Character	Integer

1. The IABS, ISIGN, IDIM, and integer MOD intrinsics accept either INTEGER*2 arguments or INTEGER*4 arguments and the result is the same type.
2. The result for MOD, AMOD, and DMOD is undefined when the value of the second argument is zero.
3. If the value of the first argument of ISIGN, SIGN, or DSIGN is zero, the result is zero.

Table A-1 (continued). Intrinsic Functions

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Index of a Substring	2	INDEX	INDEX	Character	Integer
Character	1		CHAR	Integer	Character
Character (ASCII value of 1-byte char arg)	1		ICHAR	Character	Integer
Imaginary part of cmplx#	1	IMAG	AIMAG DIMAG	Complex Complex*16	Real Double
Complex Conjugate	1	CONJG	CONJG DCONJG	Complex Complex*16	Complex Complex*16
Square Root	1	SQRT	SQRT DSQRT CSQRT ZSQRT	Real Double Complex Complex*16	Real Double Complex Complex*16
Exponential	1	EXP	EXP DEXP CEXP ZEXP	Real Double Complex Complex*16	Real Double Complex Complex*16
Natural Logarithm	1	LOG	ALOG DLOG CLOG ZLOG	Real Double Complex Complex*16	Real Double Complex Complex*16

1. 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.
2. 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.
3. 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.

Table A-1 (continued). Intrinsic Functions

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Common Logarithm	1	LOG10	ALOG10	Real Double	Real Double
Sine	1	SIN	SIN DSIN CSIN ZSIN	Real Double Complex Complex*16	Real Double Complex Complex*16
Sine (degree)	1	SIND	SIND DSIND	Real Double	Real Double
Cosine	1	COS	COS DCOS CCOS ZCOS	Real Double Complex Complex*16	Real Double Complex Complex*16
Cosine (degree)	1	COSD	COSD DCOSD	Real Double	Real Double
Tangent	1	TAN	TAN DTAN	Real Double	Real Double
Tangent (degree)	1	TAND	TAND DTAND	Real Double	Real Double
Arcsine	1	ASIN	ASIN DASIN	Real Double	Real Double
Arcsine (degree)	1	ASIND	ASIND DASIND	Real Double	Real Double
Arccosine	1	ACOS	ACOS DACOS	Real Double	Real Double
Arccsine (degree)	1	ACOSD	ACOSD DACOSD	Real Double	Real Double

1. The absolute value of the argument for ALOG10 and DLOG10 must be greater than zero.
2. The argument for SIN, DSIN, CSIN, ZSIN, COS, DCOS CCOS, ZCOS, TAN, or DTAN must be in radians and is treated modulo 2π.
3. The argument for SIND, COSD, or TAND must be in degrees and is treated as modulo 360.
4. The absolute value of the arguments for ASIN, DASIN, ASIND, DASIND, ACOS, DACOS, ACOSD, and DACOSD must be less than or equal to 1.
5. 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 for ACOS and DACOS is less than or equal to one.
6. The result of ASIN, DASIN, ACOS, and DACOS is in radians.
7. The result of ASIND, DASIND, ACOS, DACOSD is in degrees.

Table A-1 (continued). Intrinsic Functions

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Arctangent	1	ATAN	ATAN DTAN	Real Double	Real Double
Arctangent (degree)	1	ATAND	ATAND DTAND	Real Double	Real Double
Arctangent	2	ATAN2	ATAN2 DATAND2	Real Double	Real Double
Arctangent (degree)	2	ATAN2D	ATAN2D DATAN2D	Real Double	Real Double
Hyperbolic Sine	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent	1	TANH	TANH DTANH	Real Double	Real Double
Logically Greater than or Equal	2	LGE	LGE	Character	Logical
Logically Greater Than	2	LGT	LGT	Character	Logical
Logically Less than or Equal	2	LLE	LLE	Character	Logical
Logically Less Than	2	LLT	LLT	Character	Logical

1. The result of ATAN, DATAN, ATAN2, and DTAN2 is in radians.
2. The result of ATAND, DATAND, ATAN2D, and DATAN2D is in degrees.
3. 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 argument 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 absolute value of the result is P/2. Both arguments must not have the value zero.
4. 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.
5. The character relational intrinsics (LLT, LGT, LLE, and LGE) return result type LOGICAL*2 if the **$log2** (**Chapter 11**) compile option is in effect; otherwise, the result type is LOGICAL*4.

Table A-1 (continued). Intrinsic Functions

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Maximum Value	2 or more	MAX	MAX0	Integer	Integer
		MAX0	IMAX0	Integer*2	Integer*2
			JMAX0	Integer*4	Integer*4
			AMAX0	Integer	Real
		MAX1	MAX1	Real	Integer
			AMAX1	Real	Real
			DAMX1	Double	Double
			IMAX1	Real*4	Integer*2
			JMAX1	Real*4	Integer*4
		AMAX0	AIMAX0	Integer*2	Real*4
			AJMAX0	Integer*4	Real*4
Minimum Value	2 or more	MIN	MIN0	Integer	Integer
		MIN0	IMIN0	Integer*2	Integer*2
			JMIN0	Integer*4	Integer*4
			AMIN0	Integer	Real
		MIN1	MIN1	Real	Integer
			AMIN1	Real	Real
			DAMX1	Double	Double
			IMIN1	Real*4	Integer*2
			JMIN1	Real*4	Integer*4
		AMIN0	AIMIN0	Integer*2	Real*4
			AJMIN0	Integer*4	Real*4

1. The following intrinsics accept either INTEGER*2 arguments or INTEGER*4 arguments and the result is the same type as the following arguments:

IANw	ISHFT	
IOR	NOT	IEOR

When one of the intrinsic names listed in this note is specified as an *argument* in a subroutine call or function reference, the compiler supplies either an INTEGER*2 or INTEGER*4 function depending on the —**i2** command line option.

Table A-1 (continued). Intrinsic Functions

Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Bit Extraction	3		IIBITS	Integer*2	Integer*2
			JIBITS	Integer	Integer
			IBITS	Integer	Integer
Bit Set	2		IIBSET	Integer*2	Integer*2
			JIBSET	Integer	Integer
			IBSET	Integer	Integer
Bit Test	2		BITEST	Integer*2	Logical*2
			BJTEST	Integer	Logical
			BTEST	Integer	Logical
Bit Clear	2		IIBCLR	Integer*2	Integer*2
			JIBCLR	Integer	Integer
			IBCLR	Integer	Integer

Table A.1 (continued). Intrinsic Functions

Index