AIX XL FORTRAN Compiler/6000

Language Reference

SC09-1258-00

First Edition (April 1990)

This edition applies to Version 1.1 of the IBM AIX XL FORTRAN Compiler/6000 and to all subsequent releases and modifications until otherwise indicated in new editions. Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to:

IBM Canada Ltd
Information Development
Department 849
1150 Eglinton Ave East
North York, Ontario, Canada. M3C 1H7

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks and Acknowledgements

The following trademarks and acknowledgements apply to this book:

AIX is a trademark of International Business Machines Corporation.

IBM is a registered trademark of International Business Machines Corporation.

RISC System/6000 is a trademark of International Business Machines Corporation.

RT PC and RT are registered trademarks of International Business Machines Corporation.

Systems Application Architecture and SAA are trademarks of International Business Machines Corporation.

Contents

Chapter 1. Introduction

This reference manual describes the IBM AIX XL FORTRAN Compiler/6000 language. FORTRAN (FORmula TRANslation) is a high–level programming language primarily designed for applications involving numeric computations. FORTRAN is suited to most scientific, engineering, and mathematical applications.

The exceptional (XL) family of compilers provides consistency and high performance across multiple programming languages by sharing the same code optimization technology.

XL FORTRAN is a full implementation of the American National Standards Institute (ANSI) standard for FORTRAN 77 (ANSI X3.9–1978) with selected VS FORTRAN, RT PC VS FORTRAN, and RT PC FORTRAN 77 extensions.

The XL FORTRAN compiler conforms to the Systems Application Architecture (SAA) definition of the FORTRAN language.

Note: Extensions noted in Appendix C, "IBM AIX XL FORTRAN Compiler/6000 Extensions" are extensions over the language defined in *Systems Application Architecture Common Programming Interface FORTRAN Reference,* SC26–4357.

Who Should Use This Manual

Programmers using this manual should possess some knowledge of FORTRAN concepts and have previous experience in writing FORTRAN application programs.

This manual contains reference information about XL FORTRAN. It is not written as a tutorial. Therefore, if you do not have any prior FORTRAN knowledge or experience, you may wish to first obtain any of the tutorial–style FORTRAN books that are available.

How to Use This Book

This book is not intended to be a tutorial, but rather it explains the details of the XL FORTRAN language. It is designed to be used with the *User's Guide for IBM AIX XL FORTRAN Compiler/6000*, SC09–1257–00.

How to Read the Syntax Diagrams

Throughout this book, syntax diagrams use the structure defined below:

- Syntax diagrams are read from left to right and from top to bottom, following the path of the line.

  The —— symbol indicates the beginning of the diagram.

  The ——► symbol indicates that the syntax is continued on the next line.

  The ►—— symbol indicates that the syntax is continued from the previous line.

  The ——| symbol indicates the end of the diagram.

  Diagrams of syntactical units other than complete statements start with the ►—— symbol and end with the ——► symbol.

- Keywords appear in the diagrams in uppercase; for example, **OPEN**, **COMMON**, and **END**. You must spell them exactly as shown.

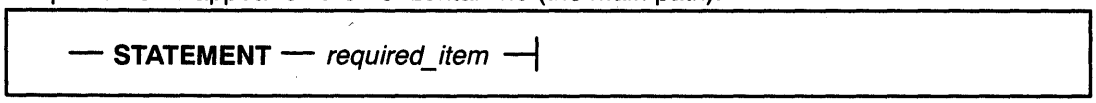
  Note: You can type keywords in uppercase, lowercase, or mixed case, and the compiler folds them into lowercase during compilation. However, if you specify the **MIXED** compiler option you must enter keywords in lowercase.

- Variables and user–supplied names appear in lowercase italics; for example, *array_element_name*. If one of these terms ends in *_list* it specifies a list of terms. A list is a nonempty sequence of the terms separated by commas. For example, the term *name_list* specifies a list of the term *name*.

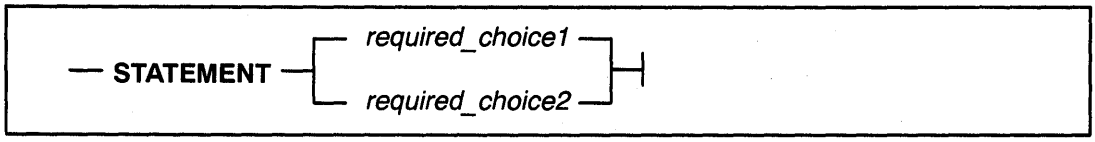
- Punctuation marks, parentheses, arithmetic operators, and other special characters must be entered as part of the syntax.

Required and Optional Items

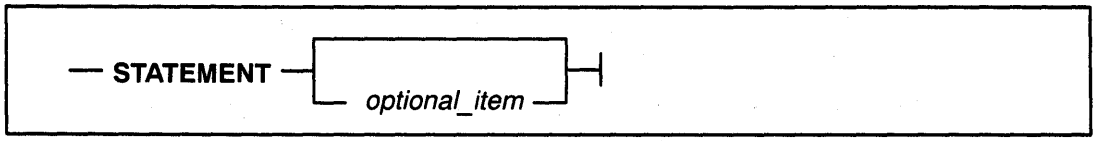
Required items appear on the horizontal line (the main path).



Branching shows two paths through the syntax.

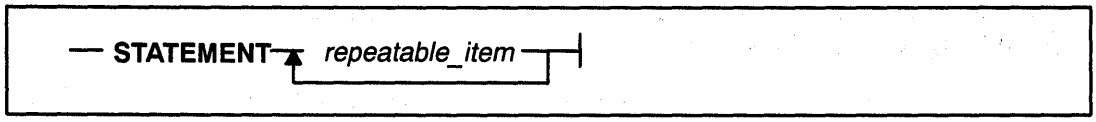


Optional items appear on the lower line of a branched path. The upper line is empty, indicating that you do not need to code anything for this syntax item.

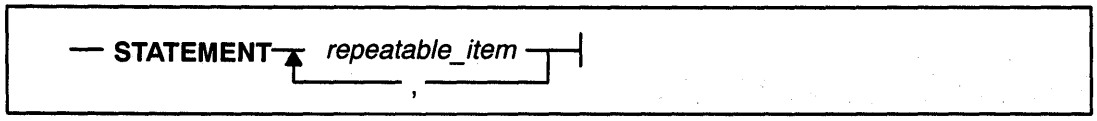


Repeatable Items

An arrow returning to the left below a line shows items that you can repeat.

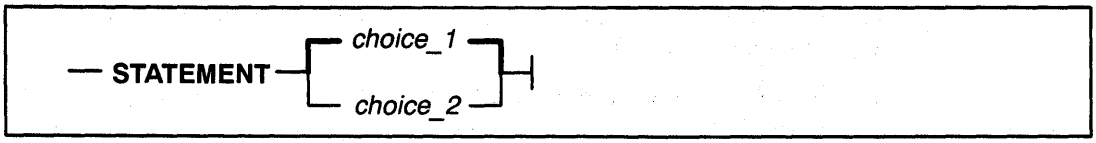


Punctuation on a repeat arrow must be placed between the repeated items.



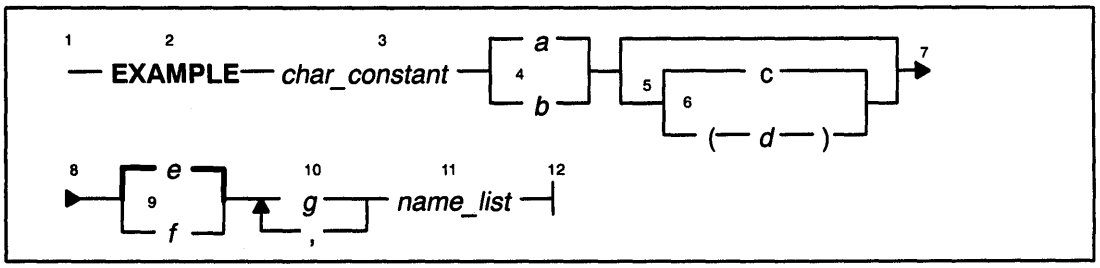
Default Items

A heavy line is the default path. Coding nothing for that item is the same as coding the default item.



Example of a Syntax Diagram

The following example of a fictitious statement shows how to use the syntax:



Interpret the diagram by following the numbers:

1. This is the start of the diagram.
2. Enter the keyword **EXAMPLE**.
3. Enter a value for *char_constant*.
4. Enter a value for *a* or *b*, but not for both.
5. This path is optional.
6. Enter a value for *c* or *d*, or no value. If you enter a value for *d*, you must include the parentheses.
7. The diagram is continued at 8.
8. The diagram is continued from 7.
9. Enter a value for *e* or *f*, or no value. If you do not enter a value, the default value *e* is used.
10. Enter at least one value for *g*. If you enter more than one value, you must put a comma between each.
11. Enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each.
12. This is the end of the diagram.

A Note About Examples

Examples in this book explain elements of the XL FORTRAN language. They are coded in a simple style. They do not try to conserve storage, check for errors, achieve fast run times, or demonstrate all possible uses of a language element.

Related Documentation

You might want to refer to the following publications for additional information:

IBM Publications:

User's Guide for IBM AIX XL FORTRAN Compiler/6000, SC09–1257–00, describes how to compile, link, and run XL FORTRAN source programs on the IBM AIX RISC System/6000 computer.

Systems Application Architecture Common Programming Interface FORTRAN Reference, SC26–4357, describes the FORTRAN component of the common programming interface.

Non–IBM Publications:

American National Standard Programming Language FORTRAN, ANSI X3.9–1978

International Standards Organization Programming Language FORTRAN, ISO 1539–1980(E)

Federal Information Processing Standards Publication FORTRAN, FIPS PUB 69

Instrument Society of America Standard: Industrial Computer System FORTRAN Procedures for Executive Functions, Process Input/Output and Bit Manipulation, ANSI/ISA S61.1

Military Standard FORTRAN, DOD Supplement to ANSI X3.9–1978, MIL–STD–1753

ANSI/IEEE Standard for Binary Floating–Point Arithmetic, ANSI/IEEE Std 754–1985.

Industry Standards

XL FORTRAN is designed according to the ANSI standard defined in the American National Standard Programming Language FORTRAN, ANSI X3.9–1978 (Full ANSI FORTRAN 77). This standard is adopted by International Standards Organization (ISO) and Federal Information Processing Standards (FIPS). Conformance to ANSI X3.9–1978 implies conformance to the following standards:

- International Standards Organization ISO 1539–1980(E), Programming Languages — FORTRAN.
- Federal Information Processing Standard, FIPS PUB 69, FORTRAN
- ANSI/IEEE Standard for Binary Floating–Point Arithmetic, ANSI/IEEE Std 754–1985.

In addition, XL FORTRAN partially complies with the FORTRAN Military Standard (MIL–STD–1753). This is a Department of Defense (DOD) Supplement to ANSI X3.9–1978. XL FORTRAN provides the following extensions to ANSI FORTRAN 77, defined in the Military Standard:

- END DO
- DO WHILE
- INCLUDE (with a different syntax)
- IMPLICIT NONE
- Bit field manipulation intrinsic functions:
  - Logical operations: IOR, IAND, NOT, IEOR
  - Shift operations: ISHFT, ISHFTC
  - Bit subfields: IBITS, MVBITS
  - Bit processing: BTEST, IBSET, IBCLR
- Octal and hexadecimal bit constants (O'd_1. . . . d_n' and Z'h_1. . . . h_n') in DATA statements.

XL FORTRAN Extensions

The XL FORTRAN language comprises:

- FORTRAN 77:

  The full ANSI FORTRAN 77 language (referred to as FORTRAN 77), defined in the document *American National Standard Programming Language FORTRAN,* ANSI X3.9–1978.

- XL FORTRAN extensions:

  These extensions are primarily, though not exclusively, selected extensions that have been commonly available in earlier IBM FORTRAN compilers, and that the VS FORTRAN, RT PC FORTRAN 77, and RT PC VS FORTRAN compilers currently support.

You can find a list of the XL FORTRAN extensions in Appendix C, "IBM AIX XL FORTRAN Compiler/6000 Extensions".

Valid and Invalid XL FORTRAN Programs

This manual defines the syntax, semantics, and restrictions you must follow when writing valid XL FORTRAN programs. The compiler discovers most violations of the XL FORTRAN language rules, but some syntactic and semantic combinations may not be found: some because they are detectable only at run time, others for performance reasons. Programs that contain these undiagnosed combinations are invalid XL FORTRAN programs, whether or not they run as expected.

Chapter 2. The Language Elements

This chapter describes the elements of an XL FORTRAN program:

- Characters
- Names
- Keywords
- XL FORTRAN statements
- Tabs
- Nonsignificant blanks
- Statement labels
- Order of statements
- Execution sequence.

Characters

The XL FORTRAN character set consists of letters, digits, and special characters:

Letters				Digits	Special Characters	
A	N	a	n	0		Blank
B	O	b	o	1	!	Exclamation point
C	P	c	p	2	"	Double quotation mark
D	Q	d	q	3	$	Currency symbol
E	R	e	r	4	%	Percent sign
F	S	f	s	5	'	Apostrophe
G	T	g	t	6	(	Left parenthesis
H	U	h	u	7	)	Right parenthesis
I	V	i	v	8	*	Asterisk
J	W	j	w	9	+	Plus sign
K	X	k	x		,	Comma
L	Y	l	y		-	Minus sign
M	Z	m	z		.	Decimal point / period
					/	Slash
					:	Colon
					<	Less than
					=	Equal sign
					>	Greater than
					_	Underscore

The characters have an order known as a collating sequence. The collating sequence is the arrangement of characters for a given system that determines their comparison status. XL FORTRAN uses ASCII (American National Standard Code for Information Interchange) to determine the ordinal sequence of characters. (See the *IBM AIX XL FORTRAN/6000 User's Guide – Appendix B* for a table of the ASCII character set.)

Names

A name (or symbolic name) is a sequence of 1 to 250 letters or digits, the first of which must be a letter. XL FORTRAN treats the currency symbol ($) and underscore character (_) as letters when you use them in a name, and you can use either as the first character. Note that the use of $ as the first character in external names can cause unpredictable results in AIX shell procedures, because AIX uses $ as the first character in a shell variable name. Also, underscore (_) is reserved for system use and some compiler generated names, so you may want to avoid using it as your first character.

XL FORTRAN folds all letters in a source program to lowercase unless they are in a character context.[1]

Note: If you specify the **MIXED** compiler option, XL FORTRAN does not fold the source program, and symbolic names are distinct if you specify them in a different case.

For example,XL FORTRAN treats

```
ia Ia iA IA
```

the same by default, but differently if you specify the **MIXED** option.

Note: If you specify the **MIXED** option, you must enter keywords in lowercase.

A name can identify the following items in a program unit:

- An array and the elements of that array
- A variable
- A constant
- A main program
- A statement function
- An intrinsic function
- A function subprogram
- A subroutine subprogram
- An entry in a function or subroutine subprogram
- A block data subprogram
- A common block
- An external user–supplied subprogram that the compiler cannot classify as either a subroutine or function subprogram name by its usage in that program unit. (See "EXTERNAL" on page 54 for more information on external subprograms.)
- A **NAMELIST** name.

The Scope of a Name

Each name in a program unit has a scope. That scope is either global to an executable program or local to a program unit, with the following exceptions:

- The name of a common block in a program unit can also be the name of an array, a statement function, a dummy procedure, a named constant, or a variable. (It cannot be a variable name that is also an external function name in a function subprogram).
- In a function subprogram, at least one function name (on the **FUNCTION** or **ENTRY** statement) must also be the name of a variable in that function subprogram.

[1] A character context means characters within character constants, Hollerith constants, format–item lists in FORMAT statements, and comments.

Names with global scope are the name of the main program, and the names of all common blocks, external functions, subroutines, and block data subprograms. All of these names have the scope of an executable program.

Names with local scope are:

- Names of variables, arrays, constants, statement functions, dummy procedures, intrinsic functions, and **NAMELIST** names. These names have a scope of a program unit. (XL FORTRAN classifies a name that is a dummy argument as a variable, array, or dummy procedure.)
- Names of variables that appear as dummy arguments in a statement function statement. These names have a scope of that statement.
- Names of variables that appear as the **DO** variable of an implied–**DO** list in a **DATA** statement. These names have a scope of the implied–**DO** list.

Keywords

A keyword is a sequence of characters that, in certain contexts, identifies a language construct. XL FORTRAN does not reserve any sequence of characters in all contexts. You can write keywords in uppercase, lowercase, or mixed case, but XL FORTRAN folds them to lowercase. If you specify the **MIXED** compiler option, the compiler does not fold the source program, and you must write keywords in lowercase.

Statements

A FORTRAN statement is a sequence of syntactic items. Statements form program units – a sequence of statements and optional comment lines that constitute a main program or subprogram. XL FORTRAN classifies each statement as either executable or nonexecutable. Executable statements specify actions. Nonexecutable statements describe attributes, arrangement and initial values of data, contain editing information, specify statement functions, classify program units, and specify entry points within subprograms.

The following table gives a list of statements, the category to which each belongs, and the chapter in which it is discussed.

Statement Group	**Statement**	**Executable or Nonexecutable**
Specification (chapter 6)	DIMENSION EQUIVALENCE COMMON Explicit Type: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, CHARACTER, LOGICAL, STATIC, AUTOMATIC IMPLICIT PARAMETER EXTERNAL INTRINSIC SAVE NAMELIST	Nonexecutable — specifies the characteristics and arrangement of data

Statement Group	Statement	Executable or Nonexecutable
DATA (chapter 7)	DATA	Nonexecutable — specifies the initial values of data
Assignment (chapter 8)	Arithmetic assignment Logical assignment Character assignment Statement label assignment (ASSIGN)	Executable — specifies actions
Control (chapter 9)	Unconditional GO TO Computed GO TO Assigned GO TO Arithmetic IF Logical IF Block IF ELSE IF, ELSE, END IF DO, DO WHILE, END DO CONTINUE STOP PAUSE END CALL (discussed in chapter 10) RETURN (discussed in chapter 10)	Executable — specifies actions
Program unit and procedure (chapter 10)	PROGRAM Statement function FUNCTION SUBROUTINE ENTRY BLOCK DATA	Nonexecutable — classifies program units, specifies statement functions, and specifies entry points within subprograms
Input/output (chapter 11)	READ WRITE PRINT OPEN CLOSE INQUIRE BACKSPACE ENDFILE REWIND	Executable — specifies actions
FORMAT (chapter 12)	FORMAT	Nonexecutable — contains editing information
Compiler directive (chapter 14)	INCLUDE EJECT @PROCESS	Neither — compiler directives

Debug statements are lines that have a D in column 1. The compiler treats these statements as comments unless you specify the **DLINES** compiler option. (See Chapter 13, "Debug Lines" for a description of these statements.)

XL FORTRAN accepts source input in either of two formats: fixed–form input format or free–form input format. You must write each statement according to the source input format specified by your selected compiler options (**FIXED** or **FREE**). The default input format is **FIXED**.

Fixed–Form Input Format

In fixed–form input format, each line is a sequence of 72 characters. Columns 73 and beyond are not significant to the compiler, and you can use them for identification, sequencing, or any other purpose.

An initial line contains a statement label, if you desire, in columns 1 to 5, a blank or zero in column position 6, and the characters representing the statement in columns 7 to 72. If there is no statement label, leave columns 1 to 5 blank.

The text of any statement except the **END** statement and the **EJECT**, **INCLUDE**, and **@PROCESS** compiler directives can continue on the following line. A continuation line contains blanks in columns 1 to 5, and any character in theXL FORTRAN character set other than a blank or zero in column position 6. XL FORTRAN allows columns 1 to 5 to contain characters, but the compiler ignores them. You can use up to 99 continuation lines for a single statement.

If you continue a Hollerith or character constant it will contain blanks from the last column position to column 72.

```
      CHARSTR="THIS IS A CONTINUED ! There will be blanks in the
     X CHARACTER STRING"           ! string between "CONTINUED" and
                                   ! "CHARACTER"
```

A comment indicator (`C`, `c`, or `*`) in column 1 will cause the compiler to treat the line as a comment line. Comment lines do not affect the executable program and you can use them to provide documentation. They can have either of two forms:

- `C`, `c`, or `*` in column 1 and, optionally, any characters you can use in a character constant in columns 2 through 72. (See "Character Constants" on page 21 for a further description.)
- Blanks in columns 1 through 72.

A comment line must not follow a line to be continued, and you cannot continue it.

Note that a D in column 1 will also cause the compiler to treat the line as a comment line if you do not specify the **DLINES** compiler option.

A comment line can appear anywhere in the program unit before the **END** statement. There is no restriction on the number of comment lines you can use.

An exclamation point (`!`) initiates an inline comment except when it appears in a character context, or if it appears in column 6 (where it is treated as a continuation character). The comment extends to the end of the source line. An **END** statement can contain a comment that you initiate with `!`. An **@PROCESS** compiler directive cannot contain an inline comment.

```
c
c This is a fixed–form example
c
      DO 10 I=1,10
        WRITE(6,*)'this is the index',I  ! with an inline comment
10    CONTINUE
```

Free–Form Input Format

In free–form input format, the first character of the statement (after a label, if there is one) must be alphabetic. The maximum length of a free–form statement is 6600 characters (equivalent to 100 fixed–form lines), excluding the continuation characters and the statement labels. The statement continuation character is a minus sign (–).

An initial line can start in any column position, and can contain (as the leftmost entry on a line) a statement label. XL FORTRAN ignores leading and imbedded blanks in a statement label.

The text of any statement, except the **END** statement and the **EJECT**, **INCLUDE**, and **@PROCESS** compiler directives can continue on the following line. You indicate a line you want continued with a minus sign terminating the line. It must be the last nonblank character that is not part of a comment. The statement text of a continuation line can start in any column position. You can have up to 99 continuation lines in a single statement.

Note that a D in column 1 will also cause the compiler to treat the line as a comment line if you do not specify the **DLINES** compiler option.

A comment line must not follow a line to be continued, and you cannot continue it. It begins with a double quotation mark (") in column 1, or is a blank line.

An exclamation point (!) initiates an inline comment except when it appears in a character context. The comment extends to the end of the source line. An **END** statement can contain an inline comment that you initiate with !. An **@PROCESS** compiler directive cannot contain an inline comment.

The minus sign for continuation must precede the ! delimiter on continuation lines. You can intersperse ! commentary with free–form source lines, but you must use the hyphen (–) to continue any line. If you want to continue a character context, you cannot follow the – signifying continuation by an inline comment.

```
"
" This is a free–form example
"
 DO 10 I=1,10
 WRITE(6,*)'this is –
      the index',I
10 CONTINUE
```

Tabs

A tab character placed anywhere in columns 1 to 6 will direct the compiler to interpret the character following as being in column 7. Therefore, you cannot tab continuation lines in fixed–form input. XL FORTRAN treats any other tab characters, except for those in a character context, as blanks.

For example, if you assume the @ is the system–generated tab character, then the code segment:

```
C@Example of tab input lines
@I=0
10@CONTINUE
```

is equivalent to:

```
C    Example of tab input lines
     I=0
10   CONTINUE
```

after resolution of the tabs.

Nonsignificant Blanks

You can position as many blanks as you want in a statement or comment to improve readability. You can even imbed blanks within keywords or names, because the compiler ignores them. If you insert blanks in character or Hollerith constants,XL FORTRAN retains them and treats them as blanks within the data.

Statement Labels

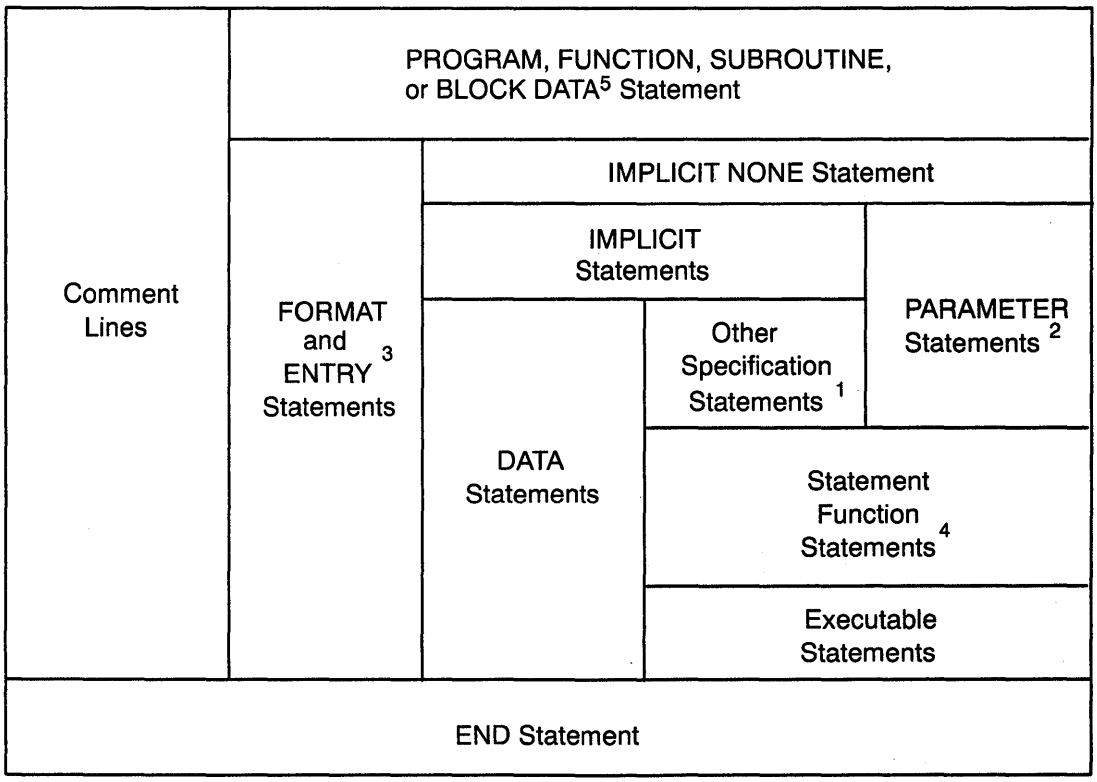
A statement label is a sequence of one to five digits, one of which must be nonzero, that you can use to identify statements in a FORTRAN program unit. You can label a fixed–form statement by placing a statement label anywhere in columns 1 through 5 of its initial line. The compiler ignores statement labels that appear on continuation lines.

Statement labels on free–format input lines must be the first nonblank characters (digits) on an initial line. You do not need blanks between the statement label and the first nonblank character following.

You must not give the same label to more than one statement in a program unit. Blanks and leading zeros are not significant in distinguishing between statement labels. You can label any statement, but you can only refer to executable statements and FORMAT statements by using statement labels. You must place the statement making the reference and the statement you want to reference in the same program unit.

Order of Statements in a Program Unit

In general, the order of statements in a program unit is given in the following diagram.



You should read the table from top to bottom to determine the order in which the statements must appear in a program unit. The vertical lines in figure delineate statements that you can intersperse. Horizontal lines delineate statements that you cannot intersperse.

The following items apply to the order of statements:

1. Explicit specification statements that initialize variables or arrays must follow other specification statements that contain the same variable or array names.
2. Any specification statement that specifies the type of a name of a constant must precede the **PARAMETER** statement that defines that name to be a constant. A **PARAMETER** statement defining the name of a constant must precede any use of the name.
3. The **ENTRY** statement cannot appear between a block **IF** statement and its corresponding **END IF** statement, or within the range of a **DO**.
4. A statement function can reference another statement function that precedes it, but not follows it.
5. The following statements are not permitted in a **BLOCK DATA** subprogram: **FORMAT**, **ENTRY**, Executable, and Statement Function Statements.

Normal Execution Sequence and Transfer of Control

Normal execution sequence is the processing of executable statements in the order in which they appear in a program unit. The normal execution sequence begins with the first executable statement in a main program. Nonexecutable statements and comment lines do not affect the normal execution sequence.

A transfer of control is an alteration of the normal execution sequence. Some statements that you can use to control the execution sequence are:

- Control statements
- The terminal statement of a **DO** loop
- Input/output statements that contain an error specifier or end–of–file specifier.

When you reference an external procedure, the execution of the program continues with the first executable statement following the **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement in the referenced procedure.

In this book, any description of the sequence of events in a specific transfer of control assumes that no event, such as the occurrence of an error or the execution of a **STOP** statement, changes that normal sequence unless otherwise specified.

Chapter 3. Data Types and Constants

This chapter describes:

- The data types
- How type is determined
- Constants and their permitted values
- The form of constants for each type.

The Data Types

A data type has a name, a set of valid values, a means to denote such values (constants), and a set of operations to manipulate the values. The following table shows each type statement with its resulting type and its associated storage length.

Type Statement	Data Type	Default Storage Length (bytes)
INTEGER*1 INTEGER*2 INTEGER*4 INTEGER	integer	1 2 4 4
REAL*4 REAL REAL*8 DOUBLE PRECISION REAL*16	real	4 4 8 8 16
COMPLEX*8 COMPLEX COMPLEX*16 DOUBLE COMPLEX COMPLEX*32	complex	8 8 16 16 32
CHARACTER CHARACTER*n	character	1 n (where 1<=n<=500)
LOGICAL*1 LOGICAL*2 LOGICAL*4 LOGICAL	logical	1 2 4 4

Note: Although you can declare extended precision data types (**REAL*16** and **COMPLEX*32**[1]), XL FORTRAN interprets them as double precision data types (**REAL*8** and **COMPLEX*16**). (See "Real" on page 17 for further explanation.)

[1]The form type*length is an abbreviation derived from the type statements. **INTEGER*2**, for example, has the same meaning as integer of length 2 bytes.

How Type Is Determined

Each variable, array, constant, expression, and function has a data type.

XL FORTRAN determines the type of a name in one of three ways:

- Explicitly, by a type statement (see "Explicit Type" on page 48 for a description of explicit type statements) or, for external functions only, by a type statement or a **FUNCTION** statement.
- Implicitly, using the **IMPLICIT** type statement. (See "IMPLICIT" on page 52 for a description of the **IMPLICIT** type statement.)
- Implicitly, by predefined convention. By default (that is, in the absence of an **IMPLICIT** type statement), if the first letter of the name is `I`, `J`, `K`, `L`, `M`, or `N`, the type is **INTEGER*4**. If the first letter of the name is any other letter (`$` and `_` included), the type is **REAL*4**.

The **IMPLICIT** type statement overrides the type as determined by the predefined convention. Explicit type statements override **IMPLICIT** statements and the predefined type specification convention.

XL FORTRAN determines the type of a constant by its form. The discussions of type in the rest of this chapter describe the form of a constant for each type.

Constants

A constant is a quantity whose value does not change. There are several classes of constants:

- Arithmetic constants
- Logical constants
- Character constants
- Hexadecimal constants
- Octal constants
- Binary constants
- Hollerith constants.

You can give a name to a constant (create a named constant) using the **PARAMETER** statement. (See "PARAMETER" on page 53 for a description of named constants.)

Arithmetic Constants

An arithmetic constant is one of:

- An integer constant
- A real constant
- A complex constant.

An arithmetic constant can be signed or unsigned:

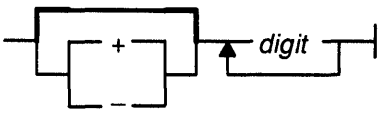
- An unsigned constant is a constant with no leading sign.
- A signed constant is a constant with a leading plus or minus sign.
- An optionally signed constant is a constant that you specify as either signed or unsigned. Only integer or real constants can be optionally signed.

XL FORTRAN considers the value zero neither positive nor negative. You can specify zero as signed or unsigned, and the value of a signed zero is not the same as the value of an unsigned zero.

Integer

An integer constant is a string of decimal digits containing no decimal point and expressing a whole number.

The form of an integer constant is:



An integer constant can be positive, zero, or negative. If unsigned and nonzero, XL FORTRAN assumes it is positive. Its magnitude must not be greater than the maximum allowed.

The following table shows the range of values that XL FORTRAN can represent using integer data types.

Data type	Length (bytes)	Range of values
INTEGER*1	1	–128 through 127
INTEGER*2	2	–32 768 through 32 767
INTEGER INTEGER*4	4	–2 147 483 648 through 2 147 483 647

If the **NOI4** compiler option has been specified, the **INTEGER** data type will have a default length of 2 bytes.

XL FORTRAN represents integers internally in two's complement notation, and the leftmost bit is the sign of the number.

Examples of Integer Constants

```
0
+91
–173
+2 147 483 647   ! largest integer value allowed
–2 147 483 648   ! smallest integer value allowed
```

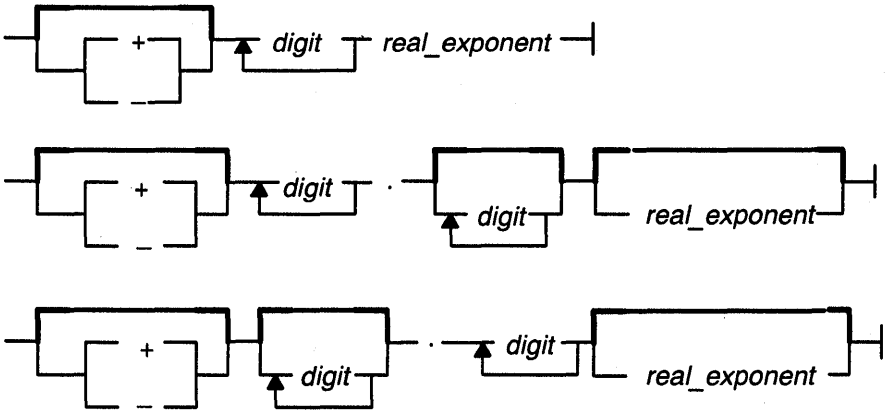
Real

The forms of a real constant are:

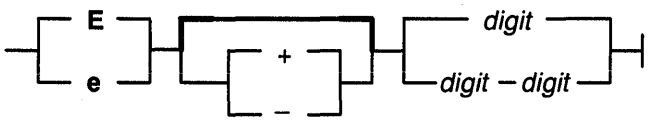
- A basic real constant
- A basic real constant followed by a real exponent
- An integer constant followed by a real exponent
- A double precision constant.

A basic real constant has an optional sign, an integer part, a decimal point, and a fractional part in that order. Both the integer part and the fractional part are strings of digits; you can omit either of these parts, but not both. You can write a basic real constant with more digits than a processor will use to approximate the value of the constant. XL FORTRAN interprets a basic real constant as a decimal number.

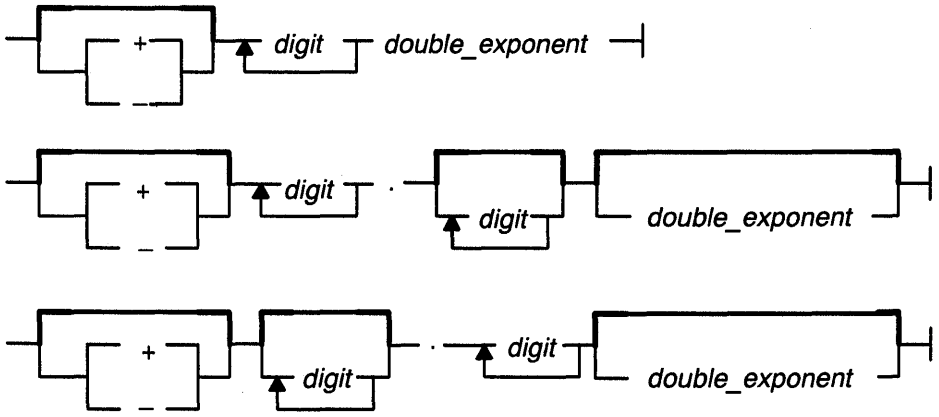
The forms of a real constant are:



A *real_exponent* is the letter E followed by a signed or unsigned one– or two–digit integer constant. It denotes a power of 10. The letter E specifies a real constant of type **REAL*4**. The form of *real_exponent* (a real exponent) is:



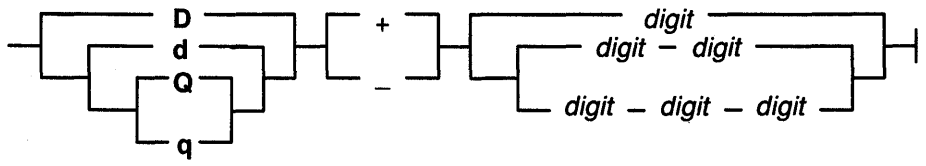
The forms of a double precision constant are:



A *double_exponent* is the letter D or Q followed by a signed or unsigned one–, two–, or three–digit integer constant. It denotes a power of 10. The letter D specifies a real constant of type **REAL*8** with length 8 bytes. The letter Q specifies a real constant of type **REAL*8** with length 16 bytes.

Note: XL FORTRAN supports the ANSI/IEEE floating–point format by allowing a three–digit integer constant to follow a double precision real constant exponent letter (that is, D or Q).

The form of *double_exponent* (a double precision exponent) is:



The following table shows the range of values that XL FORTRAN can represent with the real data types.

Data Type	Length (bytes)	Absolute Non–Zero Minimum	Absolute Maximum
REAL REAL*4	4	1.175494E–38	3.402824E+38
DOUBLE PRECISION REAL*8	8	2.225074D–308	1.797693D+308
REAL*16	16	2.225074D–308	1.797693D+308

Note: XL FORTRAN allows you to define **REAL*16** data, but it is processed as **REAL*8** data with a storage allocation of 16 bytes to preserve equivalence relationships. This convention is to allow migration of existing code to the RISC System/6000 computer. It is recommended that you use **REAL*8** for all new code. The **REAL*8** value is kept in the leftmost 8 bytes of the 16–byte space, and the rightmost 8 bytes are not used in any calculation but may be altered by data initialization or participate in unformatted input/output.

XL FORTRAN represents real numbers internally in the ANSI/IEEE binary floating–point format, which consists of a sign bit (s), a biased exponent (e), and a fraction (f).

```
bit no.   ...6....+....5....+....4....+....3....+....2....+....1....+....0
REAL*4                                  seeeeeeeeffffffffffffffffffffff
REAL*8    seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffff
```

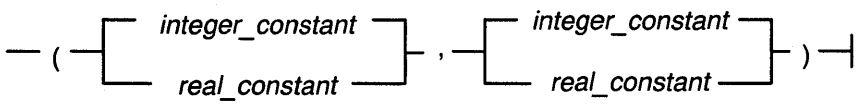
This ANSI/IEEE binary floating–point format also provides representations for +infinity, –infinity, and NaN's (not–a–number). A NaN can be further classified as a quiet NaN (NaNQ) or signalling NaN (NaNS).

Examples of Real Constants

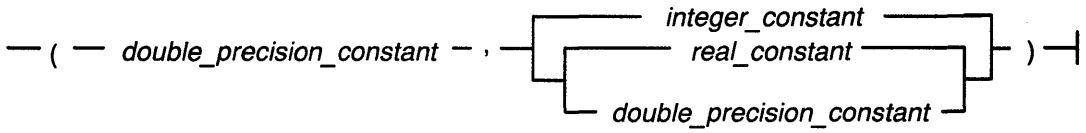
```
+0.
–999.999
7.0E+0
7.E3
1.2D+123   ! double precision constant with three–digit exponent
```

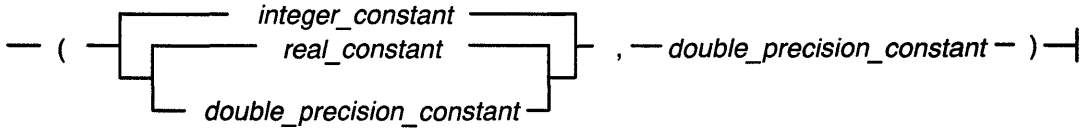
Complex

The form of a complex constant is:



The forms of a double complex constant are:





If the constants of the ordered pair representing the complex constant differ in precision, XL FORTRAN converts the constant of lower precision to a constant of the higher precision. (For example, if one constant is real and the other is double precision, the real constant is converted to a double precision constant).

If the constants of the ordered pair representing the complex constant differ in type, XL FORTRAN converts the integer constant to a real constant of the same precision as the real constant. (For example, if one constant is integer and the other is double precision, the integer constant is converted to a double precision constant).

You can also specify a complex constant using a left parenthesis followed by a pair of real or integer constant expressions (that is, expressions involving only constants or the names of constants) separated by a comma and followed by a right parenthesis.

The following table shows the range of values that XL FORTRAN can represent with the complex data types.

Data Type	Length	Absolute Non–Zero Minimum	Absolute Maximum
COMPLEX COMPLEX*8	8	(1.175494E–38, 1.175494E–38)	(3.402824E+38, 3.402824E+38)
DOUBLE COMPLEX COMPLEX*16	16	(2.225074D–308, 2.225074D–308)	(1.797693D+308, 1.797693D+308)
COMPLEX*32	32	(2.225074D–308, 2.225074D–308)	(1.797693D+308, 1.797693D+308)

Note: XL FORTRAN allows you to define **COMPLEX*32** data, but it is processed as **COMPLEX*16** data with a storage allocation of 32 bytes to preserve equivalence relationships. The **REAL*8** real part of the complex value is kept in the leftmost 8 bytes (R). The **REAL*8** imaginary part of the value (I) is located at a displacement of 16 within the item, and the remaining 16 bytes (x) are not used in any calculation but may be altered by data initialization or participate in unformatted input/output.

```
byte no. ...6....+....5....+....4....+....3....+....2....+....1....+....0
COMPLEX*32                                 RRRRRRRRxxxxxxxxIIIIIIIIxxxxxxxx
```

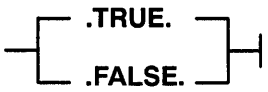
Examples of Complex Constants

```
(3,-1.86)
(-5.0E+03,.16E+02)

(-5.0E+03,.16D+02) ! This is the value (-5000.,+16.0)
                   ! and both parts are double precision

(45Q6,6D45)        ! Both parts are double precision

(1+1,2+2)          ! Use of constant expressions
```

Logical Constants

A logical constant is a constant that can have a logical value of either true or false.

The form of a logical constant is:



You can also use the abbreviations T and F (without the periods) for .TRUE. and .FALSE. respectively, but only for the initialization of logical variables or logical arrays in the DATA statement, in the explicit type statement, or in formatted input.

The following table shows the values that XL FORTRAN can represent using logical data types.

Data type	Length (bytes)	Values	Internal (hex) Representation
LOGICAL*1	1	.TRUE. .FALSE.	01 00
LOGICAL*2	2	.TRUE. .FALSE.	0001 0000
LOGICAL LOGICAL*4	4	.TRUE. .FALSE.	00000001 00000000

If the **NOI4** compiler option has been specified, the **LOGICAL** data type will have a default length of 2 bytes.

Character Constants

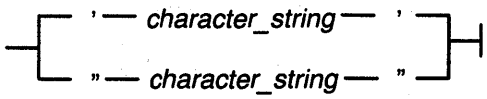
The form of a character constant is an apostrophe ('), followed by a string of characters, followed by a second apostrophe. The string can consist of any characters capable of representation in the processor, except the new–line character, because it is interpreted as the end of the source line. The delimiting apostrophes are not part of the data represented by the constant. You must represent an apostrophe within the string with two consecutive apostrophes and no intervening blanks. Blanks embedded between delimiting apostrophes are significant.

The length of a character constant is the number of characters between the delimiting apostrophes except that each pair of consecutive apostrophes counts as one character. The length of a character constant must be greater than zero. Each character requires 1 byte of storage, and XL FORTRAN uses the ASCII representation.

You can also delimit a character constant with double quotation marks (″). You must represent a double quotation mark character within a character constant delimited by double quotation marks by using two consecutive double quotation marks (without intervening blanks). The two consecutive double quotation marks count as one character.

You can place a double quotation mark character within a character constant delimited by apostrophes to represent a double quotation mark, and an apostrophe character within a character constant delimited by double quotation marks to represent a single apostrophe.

The form of a character constant is:



For compatibility with C language usage, XL FORTRAN recognizes the following backslash escapes in character strings:

Escape	Meaning
\n	New–line
\t	Tab
\b	Backspace
\f	Form feed
\0	Null
\'	Apostrophe (does not terminate a string)
\"	Double quotation mark (does not terminate a string)
\\	Backslash
\x	x, where x is any other character

All backslash characters are one character long.

The maximum length of a character constant depends upon the number of continuation lines, and the value given in the **CHARLEN** compiler option.

You can use a character constant as a data initialization value, or in any of the following:

- A character expression
- An assignment statement
- The argument list of a **CALL** statement or function reference
- An input or output statement
- A **FORMAT** statement
- A **PARAMETER** statement
- A **PAUSE** or **STOP** statement.

Examples of Character Constants

```
'0123456789'            ! Character constant of length 10
"ABCDEFGHIJ"            ! Character constant of length 10
'\"\2\'\A567\\\\\''     ! Character constant of length 10 "2'A567\\'
"\"\2\'\A567\\\\\'"     ! Character constant of length 10 "2'A567\\'
```

Note: XL FORTRAN provides support for double–byte characters within character constants, Hollerith constants, and comments. This support is provided through the **DBCS** option. Note that if the character string consists of n double–byte characters, the length of the string is 2n. Assignment of a constant containing double–byte characters to a variable, substring, or array element which is not large enough to hold the entire string may result in truncation within a double–byte character.

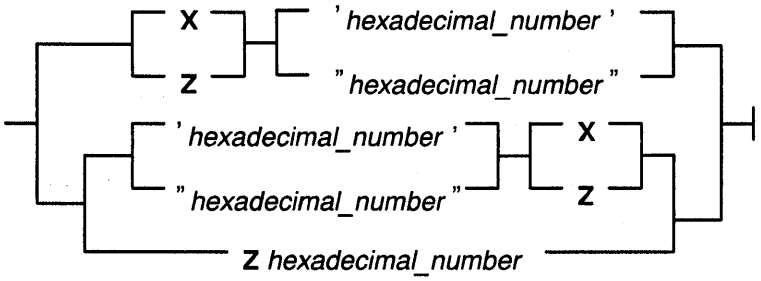
Hexadecimal Constants

The form of a hexadecimal constant is the character Z followed by a hexadecimal number formed from the digits 0 through 9 and the letters A through F or a through f, where the corresponding uppercase and lowercase letters are equivalent. You can use a hexadecimal constant of this form as a data initialization value for any type of variable or array. There is no data type corresponding to hexadecimal constants.

An alternative form of a hexadecimal constant is the character Z or the character X followed by, or preceded by, a string within apostrophes or double quotation marks. The string must be a hexadecimal number formed from the digits 0 through 9, the letters A through F, and the letters a through f. You can use a hexadecimal constant of this form in any situation where you can use a constant of any type, except as the length specification in a type declaration.

The maximum number of digits you can use in a hexadecimal constant depends upon the length specification needed for your use. If the length you have specified for the variable is x bytes, the maximum number of hexadecimal digits is 2x. If the number of digits in the hexadecimal constant is greater than the maximum,XL FORTRAN truncates the leftmost hexadecimal digits. If the number of digits is fewer than the maximum, the compiler supplies hexadecimal zeros on the left.

The form of a hexadecimal constant is:



Examples of Hexadecimal Constants

```
Z'0123456789ABCDEF'
Z"FEDCBA9876543210"
'0123456789ABCDEF'Z
Z'0123456789aBcDeF'
Z0123456789aBcDeF   !This form can only be used as an initialization
                    !value
```

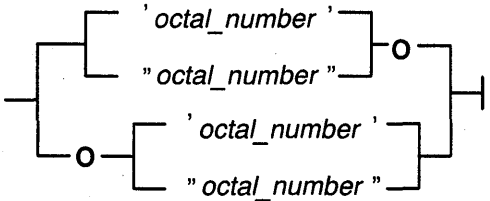
Octal Constants

The form of an octal constant is the character O followed by, or preceded by, a string within apostrophes or double quotation marks. The string must be an octal number formed from the digits 0 through 7.

The maximum number of digits allowed in an octal constant depends upon the length specification of the data object you want to represent. If the length specification of the data object is x bytes, the maximum number of octal digits is 3x (but, because an octal digit represents 3 bits, and a data object represents a multiple of 8 bits, there may be truncation of the octal constant if it consists of 3x digits). If the number of bits in the octal constant is greater than the number of bits in the data object, XL FORTRAN truncates the leftmost bits

from the octal constant. If the number of bits in the octal constant is fewer than the number of bits in the data object, the compiler supplies zero bits on the left.

You can use an octal constant in any situation where you can use a constant of any type, except as the length specification in a type declaration. The form of an octal constant is:



Examples of Octal Constants

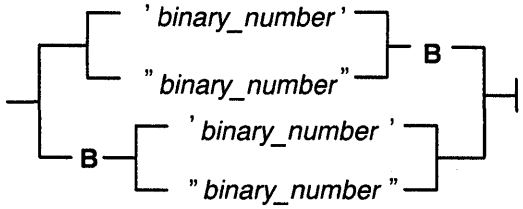
```
O'01234567'
"01234567"O
```

Binary Constants

The form of a binary constant is the character B followed by, or preceded by, a string within apostrophes or double quotation marks. The string must be a binary number formed from the digits 0 and 1.

The maximum number of digits allowed in a binary constant depends upon the length specification of the data object you want to represent. If the length specification of the data object is x bytes, the maximum number of binary digits is 8x. If the number of digits in the binary constant is greater than the number of bytes required, XL FORTRAN truncates the leftmost binary digits. If the number of digits is fewer than the number of bytes required, the compiler supplies binary zeros on the left.

You can use a binary constant in any situation where you can use a constant of any type, except as the length specification in a type declaration. The form of a binary constant is:



Examples of Binary Constants

```
B"10101010"
'10101010'B
```

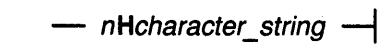
Hollerith Constants

The form of a Hollerith constant is a nonempty string of characters capable of representation in the processor and preceded by nH where n is a nonzero unsigned integer constant representing the number of characters after the H. The number of characters in the string must be greater than or equal to 1 and less than or equal to 255.

You can use a Hollerith constant in any situation where you can use a constant of any type, except as the length specification in a type declaration. If the number of characters in the Hollerith constant is greater than the maximum, XL FORTRAN truncates the rightmost

Hollerith characters. If the number of characters is fewer than the maximum, the compiler supplies blanks on the right.

The form of a Hollerith constant is:



XL FORTRAN also recognizes the backslash escapes in Hollerith constants. If a Hollerith constant contains a backslash character, n is the number of characters in the internal representation of the string, not the number of characters in the source string. (For example, you use `2H\\\\` to represent a Hollerith constant for two backslashes).

Note: XL FORTRAN provides support for double–byte characters within character constants, Hollerith constants, and comments. This support is provided through the **DBCS** option. Note that if the character string consists of `n` double–byte characters, the length of the string is `2n`. Assignment of a constant containing double–byte characters to a variable, substring, or array element which is not large enough to hold the entire string may result in truncation within a double–byte character.

Use of Hexadecimal, Octal, Binary, and Hollerith Constants

Hexadecimal, octal, binary, and Hollerith constants are "typeless" constants. They assume their data types based on the way in which you use them, and XL FORTRAN does not convert them before use.

- When you use such a constant with a unary operator, it assumes an INTEGER*4 data type.
- When you use such a constant with a binary operator, it assumes the data type of the other operand.
- When you use such a constant in a context that requires a specific data type, it assumes that data type.
- When you use such a constant as part of a complex constant, it assumes the data type of the other part of the complex constant. If both parts are typeless constants, the constants assume the real data type with length sufficient to represent both typeless constants.
- When you use such a constant as an actual argument in an external procedure reference, it assumes a length of 4 bytes but no data type.
- When you use such a constant as an actual argument in a statement function reference, it assumes the data type of the corresponding formal argument.
- When you use such a constant as an actual argument in a reference to a specific intrinsic function, it assumes the data type defined for that function.
- When you use such a constant in any other context, it assumes an INTEGER*4 data type, with the exception that a Hollerith constant in a format specification assumes a **CHARACTER** data type.

Examples of "Typeless" Constants

```
INT=B'1'           !Binary constant is integer
RL4=B'1'           !Binary constant is real
INT=INT + B'1'     !Binary constant is integer
RL4=INT + B'1'     !Binary constant is integer
INT=RL4 + B'1'     !Binary constant is real
ARRAY(B'1')=1.0    !Binary constant is integer
```

```
LOGICAL*4 LOG4
LOG4=B'1'          !Binary constant is LOGICAL*4, LOG4 is .TRUE.

LOGICAL*1 LOG1
LOG1=.NOT.B'1'     !Binary constant is LOGICAL*1, LOG1 is .FALSE.
```

Note: You can specify all "typeless" constant key letters (X, Z, O, B, and H) in uppercase or lowercase. However, if you have specified the **MIXED** compiler option, you must specify key letters in lowercase.

Chapter 4. Variables, Arrays, and Character Substrings

This chapter describes:

- Variables
- Arrays
- Character substrings
- Definition status of variables, array elements, and character substrings
- Variable, array element, and character substring references
- Association.

Variables

A variable has a name, a type, a length, and a value that can change during program execution. XL FORTRAN determines the type of a variable by the type of its name. See "How Type Is Determined" on page 16 for further explanation.

Note that an array element is not the same as a variable, as it is in some other programming languages.

Arrays

An array is an ordered, nonempty sequence of data objects. An array has a name, a type, and a sequence of values. Each element of an array has an identical length and type, and a value that may change during program execution. XL FORTRAN determines the type of an array by the type of its name. See "How Type is Determined" on page 16 for further explanation.

Array Declarators

An array declarator declares the name and size of an array. You must declare every array, and no array can have more than one array declarator for the same name. An array declarator can appear in a **DIMENSION**, **COMMON**, or explicit type statement.

The form of an array declarator is:

array_name
is the array name. Each array element has the type and length associated with this name.

dimension_declarator
A dimension declarator declares the lower and upper bounds of a dimension. Each dimension requires one dimension declarator. The minimum number of dimensions (and therefore dimension declarators) is one. The maximum number of dimensions you can have is 20.

The form of a dimension declarator is:

lower_dimension_bound
is an **INTEGER*4** arithmetic expression, called a dimension bound expression. If you do not specify this expression, a value of 1 is assumed.

upper_dimension_bound
is one of the following:

- An **INTEGER*4** arithmetic expression whose value must be greater than or equal to the value of the lower dimension bound
- An asterisk if the dimension is the last dimension in an assumed–size array declarator.

A dimension bound expression must not contain a function or an array element reference. Integer variables can appear in dimension bound expressions only in adjustable array declarators. A dimension bound expression can also be negative or zero.

Kinds of Array Declarators and Arrays

There are three kinds of array declarators:

- A constant array declarator is one in which every dimension bound expression is an integer constant expression.
- An adjustable array declarator is one in which at least one of the dimension bound expressions contains at least one integer variable name. Any variable name so used must appear either in a common block or in the same dummy argument list that contains the array name. An adjustable array declarator declares an adjustable array and its dimensions are called adjustable dimensions.
- An assumed–size array declarator is one in which the upper dimension bound of the last dimension is an asterisk.

Examples of Adjustable and Assumed–Size Array Declarators

```
SUBROUTINE MEASUR(GRIDS,LENGTH,WIDTH,TOTALS)
INTEGER LENGTH,WIDTH,TOTALS
CHARACTER*1 GRIDS(LENGTH, WIDTH) !Adjustable array
DIMENSION TOTALS(*)              !Assumed–size array
```

There are two kinds of arrays:

- An actual array is one that you declare with a constant array declarator and whose name is not a dummy argument. You can declare this kind of array in a **DIMENSION** statement, a **COMMON** statement, or a type statement.
- A dummy array is one that you can declare with constant, adjustable, or assumed–size array declarators, and whose name must be a dummy argument. You can declare this kind of array in a **DIMENSION** statement or a type statement.

Dimensions of an Array

The size of a dimension is the value of the upper dimension bound, minus the value of the lower dimension bound, plus one. XL FORTRAN does not specify the size of a dimension that has an upper dimension bound of an asterisk.

The number and size of dimensions in one array declarator can be different from the number and size of dimensions in another array declarator that you associate by common, equivalence, or argument association.

Size of an Array

The size of an array (that is, the number of elements in an array) is equal to the product of the sizes of its dimensions.

The size of an assumed–size array (declared with an asterisk) is equal to the size of its associated actual argument. For example, if the array declared in the calling routine is: `DIMENSION IARRAYA(6)`, and the associated assumed–size array is: `DIMENSION IARRAYB(*)`, the size of `IARRAYB` is 6.

Array Elements

An array consists of array elements. You identify an array element by an array element name, whose form is:

— *array_name* — (— *integer_expr_list* —) —|

array_name
is a name.

integer_expr
is an integer expression called a subscript expression.

The number of subscript expressions must be equal to the number of dimensions in the array. Subscript expressions can contain arithmetic expressions, function references that do not change any other value in the same statement, array elements, and mixed–mode expressions (integer and real only). XL FORTRAN evaluates mixed–mode expressions within a subscript according to normal FORTRAN rules. If the evaluated expression is real, the compiler converts it to integer by truncation.

The value of each subscript expression must be greater than or equal to the corresponding lower dimension bound declared for the array. The value of each subscript expression must not exceed the corresponding upper dimension bound declared for the array. If the upper dimension bound is an asterisk, the value of the corresponding subscript expression must be such that the subscript value does not exceed the size of the actual array.

The subscript value determines the element of the array that you identify by the array element name. The subscript value depends on the values of the subscript expressions and on the dimensions of the array. See "Arrangement of Arrays in Storage" below for an example.

Arrangement of Arrays in Storage

XL FORTRAN stores array elements in ascending storage units in column–major order, as in the following example of a two–dimensional array declared by array declarator `C(3,0:1)`:

	Array Element Name	Element Number
Lowest storage unit	C(1,0)	1
	C(2,0)	2
	C(3,0)	3
	C(1,1)	4
	C(2,1)	5
Highest storage unit	C(3,1)	6

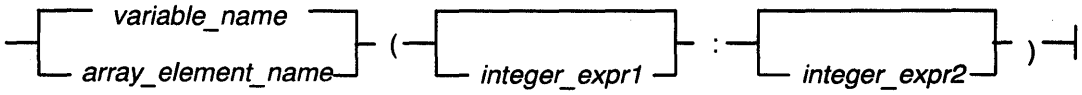
Use of Array Names

In a program unit, every appearance of an array name must be qualified by a subscript except in the following cases:

- In a dummy argument list
- In a **COMMON** statement
- In a type statement
- In an array declarator
- In an **EQUIVALENCE** statement (see "EQUIVALENCE" on page 44 for more information)
- In a **DATA** statement
- In a **SAVE** statement
- In the list of actual arguments in a reference to an external procedure
- In the list of an input/output statement if the array is not an assumed–size dummy array
- As a unit identifier for an internal file in an input/output statement (if the type of the array is character, and it is not an assumed–size array)
- As a format identifier in an input/output statement (cannot be an assumed–size array)
- In a **NAMELIST** statement.

Character Substrings

A character substring is a contiguous portion of a character variable or character array element. A character substring is identified by a substring reference whose form is:



variable_name
is the name of a character variable.

array_element_name
is the name of a character array element.

integer_expr1 and *integer_expr2*
specify the leftmost character position and rightmost character position, respectively, of the substring. Each is an integer expression called a substring expression.

The values of integer_expr1 and integer_expr2 must be such that:

1 <= integer_expr1<= integer_expr2<= length

where length is the length of the character variable or character array element. If integer_expr1 is omitted, a value of 1 is implied. If integer_expr2 is omitted, a value of length is implied.

The length of a character substring is (integer_expr2 – integer_expr1+ 1).

A substring expression can contain array element references and function references. Note that a restriction in the evaluation of expressions prohibits certain side effects. In particular, evaluation of a function must not alter the value of any other expression within the same substring name.

For the substring of a character array element, the substring information must be specified after the subscript information.

STATIC and AUTOMATIC Variables and Arrays

When a variable or array is declared **STATIC**, it indicates that there is exactly one copy of the data, and its value is retained between calls. An **AUTOMATIC** variable or array indicates that there is one copy of each variable or array for each invocation of the procedure. The default for XL FORTRAN is **STATIC**.

Variable names and array names explicitly declared as **AUTOMATIC** cannot appear in **COMMON** (as an item), **EQUIVALENCE**, **DATA**, **PARAMETER**, **SAVE**, **NAMELIST** (as an item), **EXTERNAL**, or **INTRINSIC** statements. Variable names and array names explicitly declared as **STATIC** cannot appear in **COMMON** (as an item), or **PARAMETER** statements.

The same name must not appear in more than one **AUTOMATIC** or **STATIC** type statement within the same program unit, and must not appear in both **AUTOMATIC** and **STATIC** type statements in the same program unit. Dummy argument names, statement function names, and names of constants must not appear in **AUTOMATIC** or **STATIC** type statements. Noncharacter function names can appear in **STATIC** or **AUTOMATIC** statements.

AUTOMATIC variables and arrays cannot be initialized (either with a **DATA** statement, or with an explicit type statement).

Definition Status of a Variable, Array Element, or Character Substring

At any given time during the execution of a program, the definition status of each variable, array element, or character substring is either defined or undefined:

- If defined, it has a value. The value does not change until the variable, array element, or character substring becomes undefined or until it is redefined with a different value.
- If undefined, it does not have a predictable value.

A character variable, character array element, or character substring is defined if each of its substrings of length 1 is defined. A complex variable or complex array element is defined if each of its parts is defined.

A variable, array element, or character substring must be defined at the time its value is required. A value can be assigned (thus causing definition) by:

- An assignment statement.
- An explicit type statement if initial values are provided.
- An input statement. Each variable, array element, or character substring in the input list becomes defined at the time it is assigned a value.
- Some specifiers in an input/output statement.
- A **DO** statement. The **DO** variable becomes defined.
- An input/output implied–**DO** list. The implied–**DO** variable becomes defined.
- A **DATA** statement. Initial values are provided.
- An **ASSIGN** statement.
- Association. Totally associated variables of the same type, array elements of the same type, or character substrings become defined when any one is defined. (Association is total when there is one–for–one storage mapping.)

A variable, array element, or character substring can become undefined as follows:

- All are undefined at the beginning of the program except for those specified in **DATA** statements and explicit type statements where initial values are provided.
- When a variable, array element, or character substring becomes defined, all associated variables, array elements, and character substrings of different type become undefined.
- An **ASSIGN** statement causes the specified variable to become undefined as an integer.
- If a reference to a function does not need to be evaluated to determine the value of the expression in which it appears, any variables, array elements, and character substrings in common blocks, and any arguments, that the function would have defined, become undefined.
- A **RETURN** or **END** statement causes all variables and arrays to become undefined except for the following:
  - Those in a blank common
  - Those initially defined that neither were redefined nor became undefined
  - Those specified by **SAVE** statements
  - Those specified in a named common block that appears in at least one other program unit that is either directly or indirectly referencing the subprogram.
- An error or end–of–file condition during an input statement causes all of the variables, array elements, and character substrings specified in the input list to become undefined.
- A direct access input statement that specifies a record that was not previously written causes all of the variables, array elements, and character substrings in the input list to become undefined.
- The **INQUIRE** statement may cause some variables, array elements, or substrings to become undefined. See "INQUIRE Statement" on page 109.

Reference

A variable, array element, or character substring reference is the appearance of a variable name, array element name, or character substring name in a statement in a context requiring its value to be used during program execution. When a reference is made, the current value of the variable, array element, or character substring is available. Definition of a variable, array element, or character substring is not considered a reference.

Association

Association exists if the same data item can be identified by different names in the same program unit, or by the same name or different names in different program units of the same executable program. The kinds of association are:

- Equivalence association (see page 45)
- Common association (see page 46)
- Entry association (see page 85)
- Argument association (see page 92).

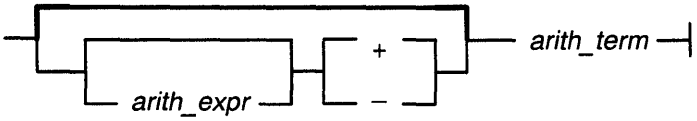
Chapter 5. Expressions

An expression, when evaluated, produces a value. This chapter describes the four kinds of expressions and explains how XL FORTRAN evaluates them:

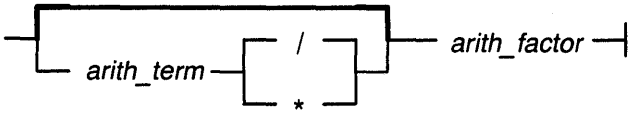
- Arithmetic
- Character
- Relational
- Logical.

Arithmetic

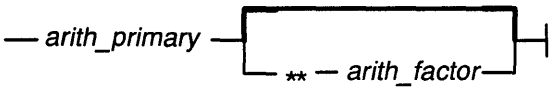
An arithmetic expression (*arith_expr*), when evaluated, produces a numeric value. The form of *arith_expr* is:



The form of *arith_term* is:



The form of *arith_factor* is:



An *arith_primary* (called a primary) is one of the following:

- An unsigned arithmetic constant
- The name of an arithmetic constant
- The name of an arithmetic variable
- The name of an arithmetic array element
- An arithmetic function reference
- An arithmetic expression enclosed in parentheses
- A hexadecimal constant
- An octal constant
- A binary constant
- A Hollerith constant.

The following table shows the available arithmetic operators and the precedence each takes within an arithmetic expression.

Arithmetic Operator	Representing	Precedence
**	Exponentiation	Highest
*	Multiplication	Intermediate
/	Division	Intermediate
+	Addition or identity	Lowest
–	Subtraction or negation	Lowest

XL FORTRAN evaluates the terms from left to right when evaluating an arithmetic expression containing two or more addition or subtraction operators. For example, `2+3+4` is the same as `(2+3)+4` after evaluation.

XL FORTRAN evaluates the factors from left to right when evaluating a term containing two or more multiplication or division operators. For example, `2*3*4` is the same as `(2*3)*4` after evaluation.

The compiler combines the primaries from right to left when evaluating a factor containing two or more exponentiation operators. For example, `2**3**4` is the same as `2**(3**4)` after evaluation.

The precedence of the operators determines the order of evaluation when XL FORTRAN is evaluating an arithmetic expression containing two or more operators having different precedence. For example, in the expression `–A**3`, the exponentiation operator (**) has precedence over the negation operator (–). Therefore, XL FORTRAN combines the operands of the exponentiation operator to form an expression that XL FORTRAN uses as the operand of the negation operator. Evaluation of the expression `–A**3` is the same as evaluation of the expression `–(A**3)`.

Note that XL FORTRAN does not allow expressions containing two consecutive arithmetic operators, such as `A**–B or A*–B`. However, you can use expressions such as `A**(–B)` and `A*(–B)`.

Examples of Arithmetic Expressions

Arithmetic Expression	Fully Parenthesized Equivalent
–b**2/2.0	–((b**2)/2.0)
i**j**2	i**(j**2)
a/b**2 – c	(a/(b**2)) – c

Arithmetic Constant Expressions

An arithmetic constant expression is an arithmetic expression in which each primary is an arithmetic constant, the name of an arithmetic constant, or an arithmetic constant expression enclosed in parentheses.

An integer constant expression is an arithmetic constant expression in which each constant or name of a constant is of type integer.

Data Type of an Arithmetic Expression

Because the identity and negation operators operate on a single operand, the type of the resulting value is the same as the type of the operand.

The following table indicates the resulting type when an arithmetic operator acts upon a pair of operands.

first operand \ second operand	I*1	I*2	I*4	R*4	R*8	X*8	X*16
I*1	I*1	I*2	I*4	R*4	R*8	X*8	X*16
I*2	I*2	I*2	I*4	R*4	R*8	X*8	X*16
I*4	I*4	I*4	I*4	R*4	R*8	X*8	X*16
R*4	R*4	R*4	R*4	R*4	R*8	X*8	X*16
R*8	R*8	R*8	R*8	R*8	R*8	X*16	X*16
X*8	X*8	X*8	X*8	X*8	X*16	X*8	X*16
X*16	X*16	X*16	X*16	X*16	X*16	X*16	X*16

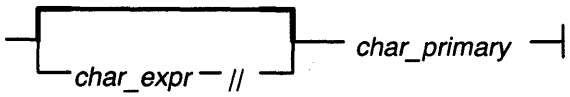
Figure 2. Data Type of an Arithmetic Expression

Notation: *T*len*, where *T* is the data type (I: integer, R: real, X: complex) and *len* is the data length in bytes.

Note that although extended precision real and complex data can be defined, it is always interpreted as double precision data, since extended precision floating–point data is not supported on the RISC System/6000 computer. Such data is allocated appropriate storage to maintain equivalence relationships.

Character

A character expression, when evaluated, produces a result of type character. The form of *char_expr* is:



char_primary (called a character primary) is one of the following:

- A character constant
- The name of a character constant
- The name of a character variable
- The name of a character array element
- The name of a character substring
- A character function reference
- A character expression enclosed in parentheses
- A hexadecimal constant
- An octal constant
- A binary constant
- A Hollerith constant.

The only character operator is //, representing concatenation.

In a character expression containing one or more concatenation operators, XL FORTRAN joins the primaries to form one string whose length is equal to the sum of the lengths of the individual primaries. For example, XL FORTRAN evaluates `'AB'//'CD'//'EF'` as `'ABCDEF'`. The length of the resulting string is six bytes.

Parentheses have no effect on the value of a character expression.

Except in a character assignment statement, a character expression must not involve concatenation of an operand whose length specifier is an asterisk in parentheses (indicating inherited length) unless the operand is the name of a constant.

Character Constant Expressions

A character constant expression is a character expression in which each character primary is a character constant, the name of a character constant, or a character constant expression enclosed in parentheses.

Example of a Character Expression

```
   CHARACTER*7  FIRSTNAME,LASTNAME
C
   FIRSTNAME='Martha'
   LASTNAME='Edwards'
   WRITE(6,*) LASTNAME//', '//FIRSTNAME  ! Output:'Edwards, Martha'
   END
```

Relational

A relational expression, when evaluated, produces a result of type logical. A relational expression can appear only within a logical expression. A relational expression can be an arithmetic relational expression or a character relational expression.

Arithmetic Relational Expressions

An arithmetic relational expression compares the values of two arithmetic expressions. Its form is:

arith_expr1
arith_expr2
are each an arithmetic expression.

relational_operator
is any of the following:

Relational Operator	Representing
.LT. or <	Less than
.LE. or <=	Less than or equal to
.EQ. or ==	Equal to
.NE. or <>	Not equal to
.GT. or >	Greater than
.GE. or >=	Greater than or equal to

If either *arith_expr1* or *arith_expr2* is of type complex, you can only specify the relational operators **.EQ.** or **.NE.**.

XL FORTRAN interprets an arithmetic relational expression as having the logical value true if the values of the operands satisfy the relation specified by the operator. If the operands do not satisfy the specified relation, the expression has the logical value false.

If *arith_expr1* and *arith_expr2* are of different data types, the value of the relational expression is the value of the expression:

((arith_expr1) – (arith_expr2)) *relational_operator* 0

where 0 is of the same type as the expression *((arith_expr1) – (arith_expr2))*.

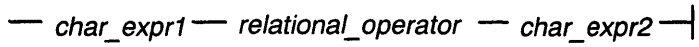
XL FORTRAN always evaluates a relational expression to a **LOGICAL*4** result, but you can convert the result in an assignment statement to a **LOGICAL*1** or **LOGICAL*2** value.

Example of an Arithmetic Relational Expression

```
IF (NODAYS .GT. 365) YEARTYPE = 'leapyear'
```

Character Relational Expressions

A character relational expression compares the values of two character expressions. Its form is:



char_expr1
char_expr2
are each a character expression.

relational_operator
is any of the relational operators described under "Arithmetic Relational Expressions" on page 36.

For operators other than **.EQ.** and **.NE.**, XL FORTRAN uses the system dependent collating sequence (determined by the ASCII coded character set) to interpret a character relational expression. The character expression whose value is lower in the collating sequence is less than the other expression. XL FORTRAN evaluates the character expressions one character at a time from left to right. You can also use the lexical intrinsic functions (**LGE**, **LGT**, **LLE**, and **LLT**) to compare character strings in ASCII order. For the operators **.EQ.** and **.NE.**, if the operands are of unequal length, the shorter will be extended on the right with blanks.

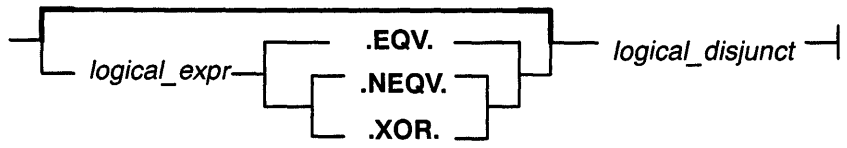
The compiler always evaluates a relational expression to a **LOGICAL*4** result, but you can convert the result to a **LOGICAL*1** or **LOGICAL*2** value in an assignment statement.

Example of a Character Relational Expression

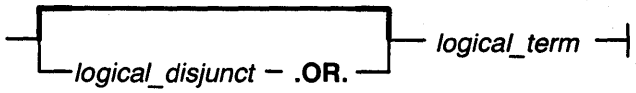
```
IF (CHARIN .GT. '0' .AND. CHARIN .LE. '9') CHAR_TYPE = 'digit'
```

Logical

A logical expression, when evaluated, produces a result of type logical. The form of a logical expression is:

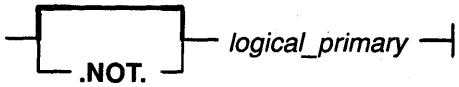


The form of a *logical_disjunct* is:



The form of a *logical_term* is:

The form of a *logical_factor* is:



logical_primary is one of the following:

- A logical constant
- The name of a logical constant
- The name of a logical variable
- The name of a logical array element
- A logical function reference
- A relational expression
- A logical expression enclosed in parentheses
- A hexadecimal constant
- An octal constant
- A binary constant
- A Hollerith constant.

The logical operators are:

Logical Operator	Representing	Precedence
.NOT.	Logical negation	First (highest)
.AND.	Logical conjunction	Second
.OR.	Logical inclusive disjunction	Third
.XOR.	Logical exclusive disjunction	Fourth (lowest)
.EQV.	Logical equivalence	Fourth (lowest)
.NEQV.	Logical nonequivalence	Fourth (lowest)

The precedence of the operators determines the order of evaluation when XL FORTRAN is evaluating a logical expression containing two or more operators having different precedence. For example, evaluation of the expression `A.OR.B.AND.C` is the same as evaluation of the expression `A.OR.(B.AND.C)`.

Value of a Logical Expression

Logical negation is evaluated as follows. Assume that `x1` and `x2` represent logical values. Then, if the value of `x1` is true, the value of **.NOT.** `x1` is false; if the value of `x1` is false, the value of **.NOT.** `x1` is true.

x1	.NOT. x1
true	false
false	true

Use the following truth table to determine the values of other logical expressions:

x1	x2	.AND.	.OR.	.XOR.	.EQV.	.NEQV.
False	False	False	False	False	True	False
False	True	False	True	True	False	True
True	False	False	True	True	False	True
True	True	True	True	False	True	False

Sometimes the compiler does not have to completely evaluate a logical expression to determine its value. Consider the following logical expression (assume that **LFCT** is a function of type logical):

```
A .LT. B .OR. LFCT(Z)
```

If `A` is less than `B`, XL FORTRAN does not have to evaluate the function reference to determine that this expression is true.

The compiler always evaluates a logical expression to a **LOGICAL*4** result, but you can convert the result to a **LOGICAL*1** or **LOGICAL*2** value in an assignment statement.

Logical Constant Expressions

A logical constant expression is a logical expression in which each primary is a logical constant, the name of a logical constant, a relational expression in which each primary is a constant expression, or a logical constant expression enclosed in parentheses.

Evaluating Expressions

Precedence of Operators

A logical expression can contain more than one kind of operator. When it does, XL FORTRAN evaluates the expression from left to right, according to the following precedence among operators:

Operator	Precedence
Arithmetic	First
Character	Second
Relational	Third
Logical	Fourth

For example, the compiler evaluates the logical expression:

```
L .OR. A + B .GE. C
```

where `L` is of type logical, and `A`, `B`, and `C` are of type real, the same as the logical expression:

```
L .OR. ((A + B) .GE. C).
```

Summary of Interpretation Rules

The following determines the order in which XL FORTRAN combines primaries using operators:

1. Use of parentheses
2. Precedence of the operators
3. Right–to–left interpretation of exponentiations in a factor
4. Left–to–right interpretation of multiplications and divisions in a term
5. Left–to–right interpretation of additions and subtractions in an arithmetic expression
6. Left–to–right interpretation of concatenations in a character expression
7. Left–to–right interpretation of conjunctions in a logical term
8. Left–to–right interpretation of disjunctions in a logical disjunct
9. Left–to–right interpretation of logical equivalences in a logical expression.

Evaluation of Expressions

The compiler evaluates arithmetic, character, relational, and logical expressions according to the following rules:

- You must define any variable, array element, function, or character substring you reference as an operand in an expression at the time the reference is made. You must define an integer operand with an integer value rather than a statement label value. Note that, if you reference a character string or substring, you must define all of the referenced characters at the time the reference is made.

  You cannot use any integer operation whose result is not mathematically defined in an executable program. (Examples are dividing by zero and raising a zero valued primary to a zero valued or negative valued power.) As well, you cannot raise a negative valued primary to a real or double precision power.

- The invocation of a function reference in a statement does not alter the value of any other entity within the statement in which the function reference appears. The invocation of a function in a statement does not alter the value of any entity in common that affects the value of any other function reference in that statement. When the value of an expression is true, invocation of a function reference in the expression of a logical IF statement affects entities in the statement. If a function reference causes definition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, you cannot use the statements:

  ```
  A(I) = F(I)

  Y = G(X) + X
  ```

  if the reference to `F` defines `I` or the reference to `G` defines `X`.

  The data type of an expression in which a function reference appears does not affect the evaluation of the actual arguments of the function, and it is not affected by the evaluation of the actual arguments of the function, except that the result of a generic function reference assumes a data type that depends on a data type of its arguments.

- An argument to a statement function reference must not be altered by evaluating that reference.

- Any occurrence of an array element reference requires the evaluation of its subscript. The data type of an expression in which a subscript appears does not affect, nor is it affected by, the evaluation of the subscript.
- Any occurrence of a substring reference requires the evaluation of its substring expressions. The data type of an expression in which a substring name appears does not affect, nor is it affected by, the evaluation of the substring expressions.

Chapter 6. Specification Statements

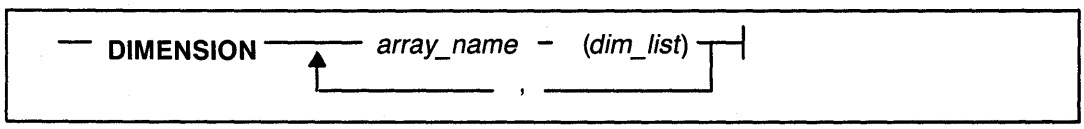
Specification statements are nonexecutable statements that describe the characteristics and arrangement of data. This chapter describes the following specification statements:

- **DIMENSION**
- **EQUIVALENCE**
- **COMMON**
- Explicit Type
- **IMPLICIT** Type
- **PARAMETER**
- **EXTERNAL**
- **INTRINSIC**
- **SAVE**
- **NAMELIST**.

Within a program unit, a name must not appear more than once in the same kind of specification statement, with these exceptions:

- A name in an **EQUIVALENCE** statement can appear more than once in the same or in different **EQUIVALENCE** statements.
- A common block name can appear more than once in the same or in different **COMMON** statements.
- A variable name or array name can appear more than once in the same or in different **NAMELIST** statements.
- A name in a **PARAMETER** statement can appear more than once in the same or in different **PARAMETER** statements provided that it is defined only once, and the rest are references to the named constant.
- The name of a constant can appear in more than one **DIMENSION**, **COMMON**, **EQUIVALENCE**, or explicit type statement provided that each appearance is a reference to the named constant.
- A name can appear in two explicit type statements if one, but not both, is an **AUTOMATIC** or **STATIC** statement.

DIMENSION



array_name
is an array name.

dim
is a dimension bound, that represents the limits for each subscript of the array in the form:

lower_dimension_bound
is an **INTEGER*4** arithmetic expression. If you do not specify this expression, the default value is 1.

upper_dimension_bound
is an **INTEGER*4** arithmetic expression that you must always specify, and it must be greater than or equal to *lower_dimension_bound*, or it is an asterisk.

(See "Arrays" on page 27 for rules about dimension bounds.)

The DIMENSION statement specifies the name and dimensions of an array. You can specify arrays with a maximum number of 20 dimensions.

Each *array_name* in a **DIMENSION** statement declares that *array_name* is an array in that program unit. You can have only one declaration of the array name as an array in a program unit. You can also declare array names and their bounds in **COMMON** statements and explicit type statements. You can specify the type of an array after it has appeared in a **DIMENSION** statement through the use of an explicit type statement. Otherwise, XL FORTRAN will implicitly define the type. The size of a dimension is equal to the upper bound minus the lower bound plus one.

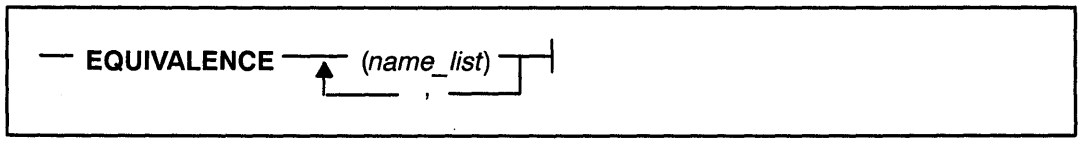
In a subprogram, if the array name is a dummy argument, the dimension bound expression can contain integer variables that must be dummy arguments or variables in a common block, and the upper bound of the last dimension can be an asterisk.

You cannot use functions or array elements in dimension bound expressions.

Examples of the DIMENSION Statement

```
DIMENSION A(10), ARRAY(5,5,5), LIST(10,100)
DIMENSION ARRAY2(1:5,1:5,1:5), LIST2(I,M)        ! adjustable array
DIMENSION B(0:24), C(-4:2), DATA(0:9,-5:4,10)
DIMENSION ARRAY (M*N*J,*)                        ! assumed-size array
```

EQUIVALENCE



name
is one of the following:

- A variable name that is not a function name.
- An array name.
- An array element name in which the subscript expressions are integer constants or names of integer constants. (You cannot use variable or function names, or array elements as subscript expressions.)
- A character substring name in which the substring expressions are integer constants or names of integer constants. (You cannot use variable or function names, or array elements.)

name must not be a dummy argument name, and you cannot declare it as **AUTOMATIC**.

The **EQUIVALENCE** statement specifies that two or more variables, arrays, array elements, or character substrings in a program unit are to share the same storage.

All items named within a pair of parentheses have the same first storage unit, and XL FORTRAN, therefore, associates them. This is called equivalence association. It may cause the association of other items as well. (See "Example 2 of an EQUIVALENCE Statement" below.) There must be at least two names within each set of parentheses.

If you specify an array element in an **EQUIVALENCE** statement, the number of subscript quantities must be equal to the number of dimensions in the array. If you specify a multi–dimensional array using an array element with a single subscript, XL FORTRAN treats the subscript as a linear element number. In all other cases, XL FORTRAN replaces the missing subscript with the lower bound of the corresponding dimension of the array. An array name without a subscript refers to the first element of the array.

For example:

```
INTEGER*4     G(2,-1:2,-3:2)
REAL*4        H(3,1:3,2:3)
EQUIVALENCE   (G(2),H(1,1))                  ! G(2) is G(2,-1,-3)
                                             ! H(1,1) is H(1,1,2)
```

Associated items can be of different data types. If so, the **EQUIVALENCE** statement does not cause type conversion. The lengths of associated items do not have to be equal.

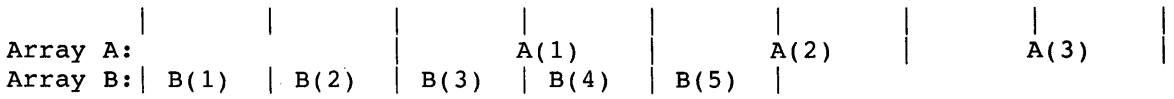
An **EQUIVALENCE** statement cannot cause the compiler to associate the storage sequences of two different common blocks in the same program unit. It must not specify that the same storage unit is to occur more than once in a storage sequence. An **EQUIVALENCE** statement must not contradict itself or any previously established associations caused by an **EQUIVALENCE** statement.

You cannot use an **EQUIVALENCE** statement with names that both appear in a common block. You can cause names not in common blocks to share storage with a name in a common block using the **EQUIVALENCE** statement. If the variable that you associate to a variable in a common block, using the **EQUIVALENCE** statement, is an element of an array, the implicit association of the rest of the elements of the array can extend the size of the common block. You cannot extend the size of the common block so that XL FORTRAN adds elements before the beginning of the established common block.

Example 1 of an EQUIVALENCE Statement

```
DOUBLE PRECISION A(3)
REAL B(5)
EQUIVALENCE (A,B(3))
```

The preceding statements associate storage units as follows:



Example 2 of an EQUIVALENCE Statement

This example shows how association of two items can result in further association. The statements:

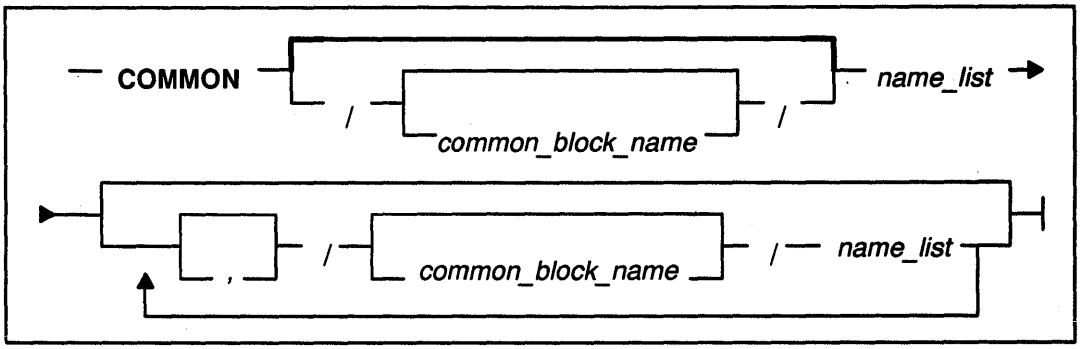
```
CHARACTER A*4,B*4,C(2)*3
EQUIVALENCE (A,C(1)),(B,C(2))
```

associate storage units as follows:

```
                |       |       |       |       |       |       |       |
Variable A:     |               A               |
Variable B:                                     |               B               |
Array C:        |       C(1)            |       C(2)            |
```

Because XL FORTRAN associates A and B with C, A and B become associated with each other.

COMMON



common_block_name
is a common block name.

name
is a variable name, array name, or array declarator. You cannot use any of these names as a dummy argument name, function name, subroutine name, entry name, or block data subprogram name. Array declarators cannot be assumed size array declarators.

The **COMMON** statement specifies common blocks and their contents. A common block is a storage area that two or more program units can share, allowing them to define and reference the same data and to share storage units.

If you specify *common_block_name*, XL FORTRAN declares all variables and arrays specified in *name_list* to be in that named common block. If you omit *common_block_name*, all variables and arrays that you specify by the following *name_list* are in a blank common block.

Within a program unit, a common block name can appear more than once in the same or in different **COMMON** statements. Each successive appearance of the same common block name continues the common block specified by that name. Common block names cannot be the same as names used in **PROGRAM**, **SUBROUTINE**, **FUNCTION**, **ENTRY**, or **BLOCK DATA** statements.

The variables and arrays in a common block can have different data types. You can mix character and noncharacter data types within the same common block. Item names in common blocks can appear in only one **COMMON** statement in a program unit, and you cannot duplicate them within the same **COMMON** statement.

Common Association

Within an executable program, all named common blocks with the same name have the same first storage unit. There can be one blank common block, and all program units that refer to blank common refer to the same first storage unit, resulting in the association of variables and arrays in different program units.

Because association is by storage unit, variables and arrays in a common block can have different names and types in different program units. Furthermore, if you use a name for a variable in one program unit, you can also use the same name for an array in another program unit.

Common Block Storage Sequence

XL FORTRAN assigns storage units for variables and arrays within a common block in the order that their names appear within the **COMMON** statement.

You can extend a common block by using an **EQUIVALENCE** statement, but only by adding beyond the last entry, not before the first entry. For example, the statements:

```
COMMON /X/ A,B
REAL C(2)
EQUIVALENCE (B,C)
```

specify common block X as follows:

```
             |   |   |   |   |   |   |   |   |   |   |   |   |
Variable A:  |       A       |
Variable B:                  |       B       |
Array C:                     |     C(1)      |     C(2)      |
```

If data is misaligned because of its specification in **COMMON**, XL FORTRAN can still process it (for example, in expressions). You can also pass misaligned data as arguments to subprograms. Note that any use of misaligned data adversely affects the performance of the program.

Size of a Common Block

The size of a common block is equal to the number of bytes of storage needed to hold all the variables and arrays in the common block, including any extensions resulting from equivalence association.

Differences between Named Common Blocks and Blank Common Blocks

The differences between named common blocks and blank common blocks are:

- Within an executable program, there can be more than one named common block but only one blank common block.
- In all program units of an executable program, named common blocks of the same name must have the same size, but blank common blocks can have different sizes. (If you specify blank common blocks with different sizes in different program units, the length of the longest block becomes the length of the blank common block in the executable program.)
- You can initially define variables and array elements in a named common block by using the **DATA** statement or by the explicit type specification statement in a **BLOCK DATA** subprogram. You cannot initially define variables and array elements in a blank common block.
- Executing a **RETURN** or **END** statement may cause entities in named common blocks to become undefined but never causes entities in blank common to become undefined.

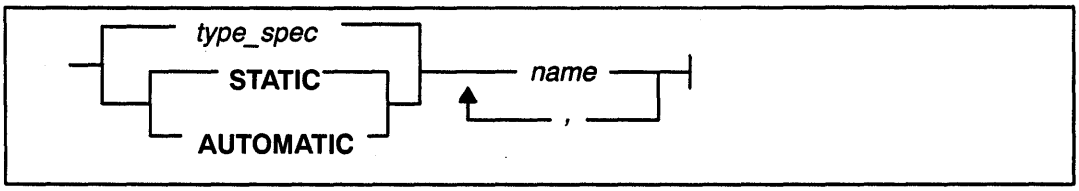
Restriction on Common and Equivalence

An **EQUIVALENCE** statement cannot cause the storage sequences of two different common blocks to become associated.

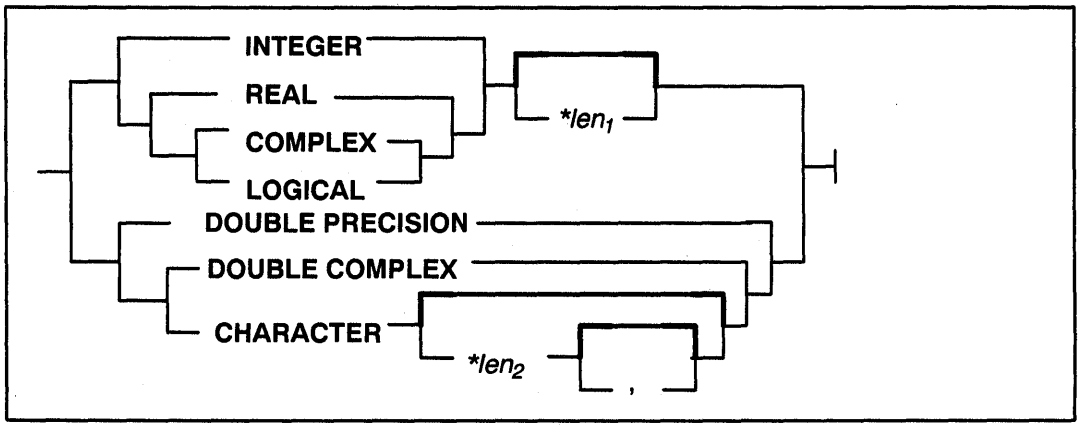
Examples of the COMMON Statement

```
INTEGER MONTH,DAY,YEAR
COMMON /DATE/ MONTH,DAY,YEAR
REAL*4          R4
REAL*8          R8
CHARACTER*1     C1
COMMON /NOLIGN/ R8,C1,R4      !R4 will not be aligned on a
                              !full-word boundary.
```

Explicit Type



type_spec
specifies a data type and is one of the following:



where:

DOUBLE COMPLEX
defines double precision complex data. It is an alternative form of **COMPLEX*16**.

STATIC indicates that there is exactly one copy of the data, and XL FORTRAN retains its value between calls. Local variables are **STATIC** by default.

AUTOMATIC indicates that there is one copy of each variable declared **AUTOMATIC** for each invocation of the procedure.

and where:

len1 is optional and *len1* represents one of the permissible length specifications for its associated type *type_spec*, as described in "The Data Types" on page 15.

The length *len1* can be an unsigned, nonzero, integer constant.

len2 specifies the length (number of characters between 1 and 32767). It is optional.

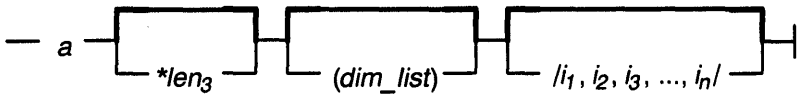
Note: You can specify the **CHARLEN** compiler option to set the maximum length of the character data type to a range of 1 through 32767. The default maximum length is 500 characters.

You can express the length **len2* as:

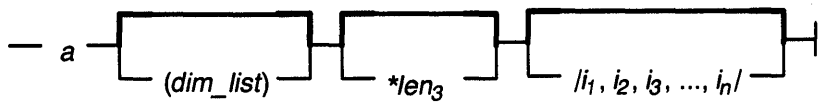
- An unsigned, nonzero, integer constant
- An expression with a positive value that contains integer constants enclosed in parentheses
- An asterisk enclosed in parentheses.

XL FORTRAN uses the length **len2* immediately following the word **CHARACTER** as the length specification of any name in that statement that has no length specification. To override a length for a particular name, see the alternative forms of *name* below. If you do not specify **len2*, its default value is 1.

name
is a variable, array, function name, or the name of a constant. It can have the form:



or



where:

a is variable, array, function name, dummy procedure name, or named constant.

dim_list is a list of dimension bounds (separated by commas) that you can only specify for arrays. They represent the limits for each subscript of the array in the form:

lower_dimension_bound
is an **INTEGER*4** arithmetic expression. If you do not specify this expression, its default value is 1.

upper_dimension_bound
is either an **INTEGER*4** arithmetic expression that is greater than or equal to lower_dimension_bound, or is an asterisk. You must always specify it.

(See "Arrays" on page 27 for rules about dimension bounds.)

****len3*** overrides the length as specified in the **CHARACTER** statement, or as specified in the initial keyword of the statement.

i1, i2, i3 . . . in are optional and represent initial data values.

The explicit type statement:

- Specifies the type and length of variables, arrays, named constants, and user–supplied functions
- Specifies the dimensions of an array
- Assigns initial data values for variables and arrays.

The explicit type statement overrides the **IMPLICIT** statement, which, in turn, overrides the predefined convention for specifying type.

You can use an explicit type statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in an explicit type specification statement does not cause the name to lose its intrinsic function property. However, if you define the name as an array, or subsequently use it as a variable (for example, in an assignment statement), it loses its intrinsic function property. Also, if you assign an initial value to the name, it loses its intrinsic function property.

Function and array element names must not appear in an expression for the length specification. The value of an expression for a length specification must be a positive, nonzero, integer constant. Any length assigned must be an allowable value for the associated data type. The length specified (or assigned by default) with an array name is the length of each element of the array.

If the **CHARACTER** statement is in a main program, and you specify the length of *name* as an asterisk enclosed in parentheses (*) (also known as inherited length), the *name* must be the name of a character constant defined in a **PARAMETER** statement. The character constant assumes the length of its corresponding expression in the **PARAMETER** statement.

If the **CHARACTER** statement is in a subroutine subprogram, and the length of *name* is inherited, *name* must be the name of a dummy argument or the name of a character constant defined in a **PARAMETER** statement. If *name* is the name of a dummy argument, the dummy argument assumes the length of the associated actual argument for each reference to the subroutine. If *name* is the name of a character constant, the character constant assumes the length of its corresponding expression in the **PARAMETER** statement.

If the **CHARACTER** statement is in a function subprogram, and the length of *name* is inherited, *name* must be either the name of a dummy argument, the name of the function in a **FUNCTION** or **ENTRY** statement in the same program unit, or the name of a character constant defined in a **PARAMETER** statement. If *name* is the name of a dummy argument, the dummy argument assumes the length of the associated actual argument for each reference to the function. If *name* is the function name or an entry name, *name* assumes the length specified in the calling program unit. If *name* is the name of a character constant, the character constant assumes the length of its corresponding expression in the **PARAMETER** statement.

You cannot specify the length of a statement function of character type by an asterisk enclosed in parentheses (*), but you can specify an integer constant expression.

The length specified for a character function in a main program unit that refers to the function must be an expression involving only integer constants or names of integer constants. This length must agree with the length specified in the subprogram that specifies the function, if you do not specify the length in the subprogram as an asterisk enclosed in parentheses (*).

You cannot assign initial values to dummy arguments, names of constants, function names, statement functions, and dummy procedure names. You cannot assign initial data to items in a blank common. For a named common, you can only have initial data in a **BLOCK DATA** subprogram. You can assign initial data values to variables or arrays using i_j $(1 \leq j \leq n)$, where i_j is a constant or list of constants separated by commas. Each i_j provides initialization only for the immediately preceding variable or array. You use lists of constants to assign initial values to array elements.

If you assign initial data values to an array in an explicit specification statement, the dimension information for the array must be in the explicit specification statement or in a preceding **DIMENSION** or **COMMON** statement.

For **CHARACTER** and **LOGICAL** type statements, the data must be of the same type as the variable or array. For a numeric type statement, the data need not be of the same type as the variable or array, but must be compatible with the type of the variable. (For example, `integer i/'.TRUE.'/` is invalid.) If it is not of the same type, XL FORTRAN converts the data to the specified type. You can represent successive occurrences of the same constant with the form *i*`*`*constant*, where *i* is the repetition count, as in the **DATA** statement.

You can use a hexadecimal constant, octal constant, binary constant, or Hollerith constant to initialize a variable, array element or array of any type, or a character substring. If the hexadecimal constant is the same string as the name of a constant you defined with a previous **PARAMETER** statement, XL FORTRAN recognizes the string as the named constant, and will assign its value to the item in the explicit type statement.

If the hexadecimal, octal, or binary constant is smaller than the length (in bytes) of the variable to be initialized, XL FORTRAN adds zeros on the left. If the constant is larger, the compiler truncates the leftmost hexadecimal, octal, or binary digits. If the Hollerith constant is smaller than the length (in bytes) of the variable to be initialized, XL FORTRAN adds blanks on the right. If the constant is larger, the compiler truncates the rightmost Hollerith characters.

If a typeless constant initializes a complex data type, you use one constant that initializes both the real and the imaginary parts, and do not enclose the constant in parentheses.

You must initialize list items of type logical with logical constants, but you can use the abbreviated forms:

- T for .TRUE.
- F for .FALSE.

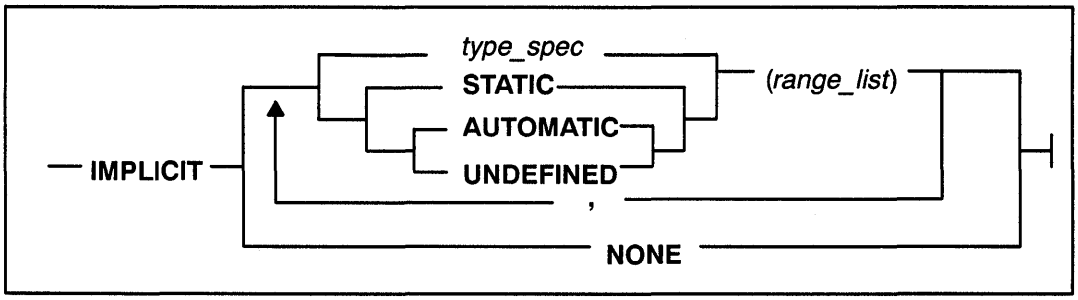
If you define the abbreviated form of a logical constant as the name of a constant in a previous **PARAMETER** statement, and if it occurs as a constant in an explicit type statement, it is no longer an abbreviated logical constant, but the name of the constant.

The table in "The Data Types" on page15 lists all of the possible explicit type specifiers, and the resulting type and length of the data item.

Examples of Explicit Type Statements

```
CHARACTER*8 ORANGES
DATA        ORANGES /'ORANGES '/

SUBROUTINE  TEST(VARBL)
CHARACTER*(*)  VARBL

REAL        TAN /1.2/       ! TAN loses its intrinsic property

COMPLEX     C,D /(2.1,4.7)/, E*16
INTEGER*2   ITEM/76/, VALUE
REAL        A(5,5)/20*6.9E2,5*1.0/, B(100)/100*0.0/
REAL*8      MASTER, HOLD, VALUE*4, ITEM(5,5)
INTEGER*1   BIN_ITEM /B'00001010'/, OCT_ITEM /O'12'/

STATIC      VAR_NAME /1.0/    ! STATIC data can be initialized
```

IMPLICIT



type_spec
specifies a data type. The description is in "Explicit Type" on page 48. You can also specify the type as **UNDEFINED**.

range
is either a single letter or a range of letters drawn from the XL FORTRAN character set in alphabetic order. (See "Characters" on page 7.) A range of letters has the form *letter*$_1$–*letter*$_2$, where *letter*$_1$ is the first letter in the range and *letter*$_2$ is the last letter in the range. Specifying a range of letters has the same effect as specifying a list of all letters in that range. The underscore (_) follows the currency symbol ($), which follows the Z. Thus, the range Y – _ is the same as Y, Z, $, _.

The **IMPLICIT** statement changes or confirms default implicit typing or, with the form **IMPLICIT NONE** specified, voids the implicit type rules altogether. See "How Type is Determined" on page16 for a discussion of the implicit rules.

For all variables, arrays, and function subprogram names that begin with the letter or letters specified by *range_list*, and for which you do not explicitly specify a type, XL FORTRAN gives the type specified by the immediately preceding *type_spec*. A letter or a range of letters that you specify as **STATIC** or **AUTOMATIC** can also appear in an **IMPLICIT** statement for any data type.

If you specify the form **IMPLICIT NONE**, you must use explicit type statements to specify data types. The compiler will issue a diagnostic message for each symbolic name that you use that does not have an explicitly defined data type. When **IMPLICIT NONE** is specified, you cannot specify any other **IMPLICIT** statement in the same program unit, except one containing **STATIC** or **AUTOMATIC**. You can also compile your program with the **UNDEF** compiler option to achieve the same effect as an **IMPLICIT NONE** statement.

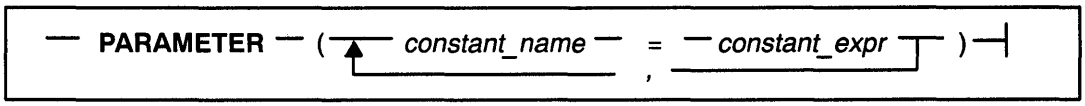
IMPLICIT UNDEFINED turns off the implicit data typing defaults for the character or range of characters specified. When you specify **IMPLICIT UNDEFINED**, you must declare the data types of all symbolic names in the program unit that start with a specified character. The compiler will issue a diagnostic message for each symbolic name that you use that does not have an explicitly defined data type.

An **IMPLICIT** statement does not change the data type of an intrinsic function.

Example of the IMPLICIT Statement

```
IMPLICIT INTEGER (B), COMPLEX (D, K–M), REAL (R–A)
```

PARAMETER



constant_name
is the name of a named constant in this program unit.

constant_expr
is a constant or a constant expression of integer, real, complex, logical, or character type.

The **PARAMETER** statement specifies names for constants. **PARAMETER** statements must appear before the use of the named constant. You must define the name only once in a **PARAMETER** statement of the program unit. Items appearing in a **PARAMETER** statement cannot appear in a **STATIC**, **AUTOMATIC**, or **COMMON** statement.

The type of the name must be defined before its definition in this statement (by **IMPLICIT**, explicit type, or predefined convention). You must also previously define the length of a character name by **IMPLICIT** or explicit type prior to its use in a **PARAMETER** statement. You cannot change the type and length of a name of a constant by subsequent specification statements, including **IMPLICIT** statements. If a **CHARACTER** name has length defined as (*), it then assumes the length of the result of the expression.

If the name (*constant_name*) is of character type, the corresponding expression (*constant_expr*) must be a character expression containing only character constants or names of character constants.

If the name (*constant_name*) is of logical type, the corresponding expression (*constant_expr*) must be a logical expression containing only logical constants or names of logical constants, or a relational expression containing only constants or names of constants.

If the name (*constant_name*) is of type integer, real, or complex, the corresponding expression (*constant_expr*) must be an integer, real, or complex expression containing only constants of these types or names of such constants. Type conversion for arithmetic constants takes place across the equal sign.

The constant expression cannot be a hexadecimal constant of the form Znn...nnn or an abbreviated logical constant, `T` or `F`. It can be a hexadecimal constant of the form `X'nn...nn'`, `Z'nn...nn'`, `X"nn...nn"`, `Z"nn...nn"`, `'nn...nn'X`, `'nn...nn'Z`, `"nn...nn"X`, or `"nn...nn"Z`, an octal constant, Hollerith constant, or a binary constant.

If the hexadecimal, octal, or binary constant is smaller than the length (in bytes) of the *constant_name* to be initialized, XL FORTRAN adds zeros on the left. If the constant is larger, the compiler truncates the leftmost hexadecimal, octal, or binary digits. If the Hollerith constant is smaller than the length (in bytes) of the *constant_name* to be initialized, XL FORTRAN adds blanks on the right. If the constant is larger, the compiler truncates the rightmost Hollerith characters.

The constant expression can contain the following intrinsic functions with constant arguments:

- ABS (for integer and real), DABS, IABS, QABS
- IMAG, AIMAG, DIMAG, QIMAG
- MAX, MAX0, AMAX0, MAX1, AMAX1, DMAX1, QMAX1
- MIN, MIN0, AMIN0, MIN1, AMIN1, DMIN1, QMIN1

- AND, IAND, OR, IOR, XOR, IEOR, NOT
- BTEST, IBSET, IBCLR, ISHFT, LSHIFT, RSHIFT
- CONJG, DCONJG, QCONJG
- All conversion intrinsic functions
- DPROD
- SIGN, ISIGN, DSIGN
- LEN.

You cannot use names of constants in a **FORMAT** statement or as statement labels. You can use them as part of a **COMPLEX** constant.

Examples of the PARAMETER Statement

```
PARAMETER  (TWO=2.0)        ! Use of (literal) constant.
PARAMETER  (TWICE=TWO)      ! Use of symbolic constant.
COMPLEX    XCONST
REAL       RPART,IPART
PARAMETER  (RPART=1.1,IPART=2.2)
PARAMETER  (XCONST = (RPART,IPART+3.3))
PARAMETER  (MAX10 = MAX(10,5))     ! use of intrinsic function
CHARACTER*2 BB
PARAMETER  (BB=Z'2020')     ! initialize to blanks
```

EXTERNAL

name

is the name of an external procedure, dummy procedure, or **BLOCK DATA** subprogram.

The **EXTERNAL** statement identifies a symbolic name as an external procedure so that you can use the name as an actual argument. The **EXTERNAL** statement must proceed all statement function definitions and executable statements.

An external procedure name must appear in an **EXTERNAL** statement in any program unit that uses the name as an actual argument. The name of any subprogram that you pass as an argument to another subprogram must appear in an **EXTERNAL** or **INTRINSIC** statement in the calling program unit. The same name cannot appear in both **EXTERNAL** and **INTRINSIC** statements. You can have only one appearance of a symbolic name in all of the **EXTERNAL** statements of a program unit.

If an intrinsic function name appears in an **EXTERNAL** statement in a program unit, the name becomes the name of a user–defined external procedure. Therefore, you cannot invoke an intrinsic function of the same name from that program unit.

A statement function name cannot appear in an **EXTERNAL** statement.

Example of the EXTERNAL Statement

```
PROGRAM MAIN
EXTERNAL TREES
CALL SAM(TREES)
END
```

```
SUBROUTINE SAM(ARG)
CALL ARG()                    !This results in a call to TREES.
END
```

INTRINSIC

name
is the name of a FORTRAN intrinsic function described in Appendix A, "Intrinsic Functions".

The **INTRINSIC** statement identifies a name as an intrinsic function and allows you to use specific names of intrinsic functions as actual arguments. The **INTRINSIC** statement is a specification statement and must precede statement function definitions and executable statements.

If you use a specific intrinsic function name as an actual argument in a program unit, it must appear in an **INTRINSIC** statement in that program unit. Generic names can be in the **INTRINSIC** statement, but you cannot pass them as arguments unless they are also specific names.

The intrinsic function names that you must not use as actual arguments are those for:

- Type conversion: **INT**, **IFIX**, **IDINT**, **HFIX**, **REAL**, **FLOAT**, **DFLOAT**, **SNGL**, **DREAL**, **DBLE**, **CMPLX**, **DCMPLX**
- Choosing largest and smallest values: **MAX**, **MAX0**, **AMAX1**, **DMAX1**, **QMAX1**, **AMAX0**, **MAX1**, **MIN**, **MIN0**, **AMIN1**, **DMIN1**, **QMIN1**, **AMIN0**, **MIN1**.

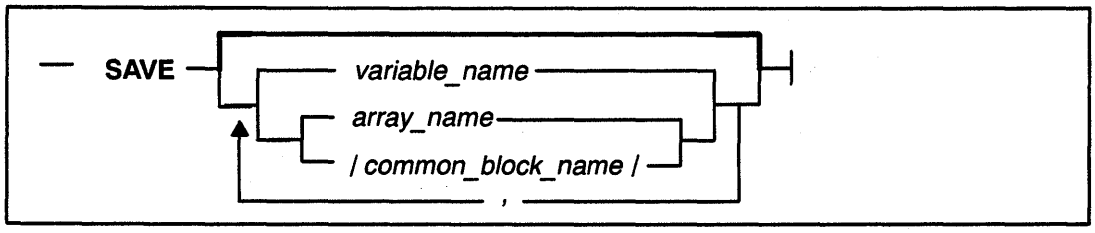
A generic or specific function named in an **INTRINSIC** statement keeps its generic or specific properties.

A name must not appear in both an **EXTERNAL** and an **INTRINSIC** statement in the same program unit. You cannot use duplicate names in **INTRINSIC** statements.

Example of the INTRINSIC Statement

```
      INTRINSIC SIN
      .
      .
c Pass intrinsic function name to subroutine
      CALL DOIT(0.5,SIN,X)
```

SAVE



variable_name, array_name, common_block_name
is the name of a variable, array, or named common block. You cannot use **AUTOMATIC** variables, dummy argument names, procedure names, entry names, or the names of variables and arrays in a common block.

The **SAVE** statement specifies the names of variables, arrays, and named common blocks whose definition status you want to retain after control returns (after executing a **RETURN** or **END**) from the subprogram in which you define the variables, arrays, and named common blocks.

XL FORTRAN treats a **SAVE** statement without a list as though it contains the names of all common items and local variables in the program unit. The appearance of a common block name in a **SAVE** statement has the effect of specifying all the entities in that named common block. You cannot use the same name twice.

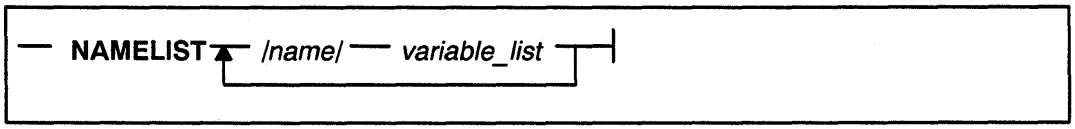
Within a function or subroutine subprogram, a variable or array whose name you specify in a **SAVE** statement does not become undefined as a result of a **RETURN** or **END** statement in the subprogram. A variable or array in a common block may become undefined or redefined in another program unit, even though you specify the common block in a **SAVE** statement.

If a local entity specified in a **SAVE** statement (and not in a common block) is in a defined state at the time XL FORTRAN executes a **RETURN** or **END** statement in a subprogram, that entity is defined with the same value at the next reference of that subprogram.

A **RETURN** statement or an **END** statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

- Entities specified by **SAVE** statements
- Entities in blank common blocks
- Initially defined entities that you do not redefine and do not become undefined
- Entities in named common blocks that appear in the subprogram and appear in at least one other program unit that is referring, either directly or indirectly, to that subprogram. The entities in a named common block may become undefined by executing a **RETURN** or **END** statement in another program unit.

NAMELIST



name
is a **NAMELIST** name. It is a name, enclosed in slashes, that must not be the same as a variable or array name, a named constant, or a procedure.

variable
is a variable name or array name.

The **NAMELIST** statement specifies one or more lists of names for use in **READ**, **WRITE**, and **PRINT** statements. The list of names belonging to a **NAMELIST** name ends with the appearance of another **NAMELIST** name or the end of the **NAMELIST** statement. The **NAMELIST** statement must precede any statement function definitions and all executable statements.

A variable name or an array name can belong to one or more **NAMELIST** lists. Neither a dummy variable nor a dummy array name can appear in a **NAMELIST** list. You must declare a **NAMELIST** name in a **NAMELIST** statement, and you can declare it only once. You cannot have subscripts or substrings in a list.

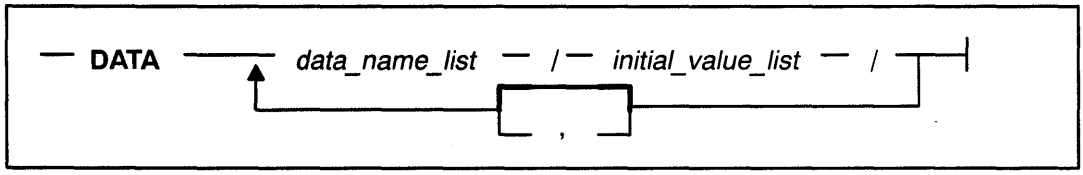
A **NAMELIST** name can appear only in input/output statements. The rules for input/output conversion of **NAMELIST** data are the same as the rules for data conversion.

See "NAMELIST Input Data" on page 136 for a description of **NAMELIST** input data.

Chapter 7. DATA Statement

This chapter describes the **DATA** statement and the use of an implied **DO** with a **DATA** statement. The **DATA** statement is a nonexecutable statement that provides initial values for variables, array elements, arrays, and character substrings.

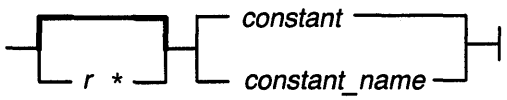
DATA Statement



data_name
is any of the following:

- A variable name
- An array name
- An array element name in which the subscript expressions are integer constant expressions
- A character substring name in which the substring expressions are integer constant expressions
- An implied–DO list.

initial_value
has the form:



r
is a nonzero, unsigned, integer constant or the name of such a constant. The form *r*constant* or *r*constant_name* is equivalent to *r* successive appearances of the constant or name of constant.

Each *data_name_list* must specify the same number of items as its corresponding *initial_value_list*. There is a one–to–one correspondence between the items in these two lists. This correspondence establishes the initial value of each *data_name*.

Specifying an array name as a *data_name* is the same as specifying a list of all the elements in the array in the order XL FORTRAN stored them. Again, there must be a one–to–one correspondence between the number of items in the array and the number of initial values.

The data type of each *data_name* and the data type of its corresponding *initial_value* must agree if either is of type logical or character. For type character, definition proceeds according to the rules for character assignment statements. (See "Character Assignment" on page 66.) If a *data_name* is of type integer, real, or complex, its corresponding *initial_value* need not agree in type, although it must be one of the types in that group (integer, real, or complex). If the types do not agree, XL FORTRAN converts the *initial_value* according to the rules for arithmetic conversion in the figure on page 63.

You can use a hexadecimal constant (of the form `Znn...nn`, `X'nn...nn'`, `Z'nn...nn'`, `X"nn...nn"`, `Z"nn...nn"`, `'nn...nn'X`, `'nn...nn'Z`, `"nn...nn"X`, or `"nn...nn"Z`), octal constant, binary constant, or Hollerith constant to initialize a variable, an array element, or an array of any type. You can also use a "typeless" constant to initialize a substring.

If the hexadecimal, octal, or binary constant is smaller than the length (in bytes) of the variable, array element, or substring to be initialized, XL FORTRAN adds zeros on the left. If the constant is larger, the compiler truncates the leftmost hexadecimal, octal, or binary digits. If the Hollerith constant is smaller than the length (in bytes) of the variable to be initialized, XL FORTRAN adds blanks on the right. If the constant is larger, the compiler truncates the rightmost Hollerith characters.

If a typeless constant initializes a complex data type, you use one constant that initializes both the real and the imaginary parts, and do not enclose the constant in parentheses.

You must initialize list items of type logical with logical constants, but you can use the abbreviated forms (`T` for `.TRUE.` and `F` for `.FALSE.`). If the abbreviated form of the logical constant is the same string as the name of a constant you defined previously in a **PARAMETER** statement, XL FORTRAN recognizes the string as the named constant and will assign its value to the corresponding list item in the **DATA** statement.

In a block data subprogram, you can use **DATA** statements or explicit type statements to provide an initial value for a variable, array element, or character substring in a named common block.

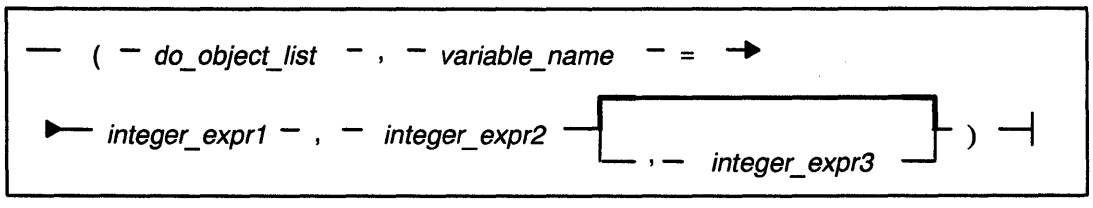
A **DATA** statement cannot provide an initial value for:

- A dummy argument
- A variable, array element, or character substring in a blank common block, including a variable, array element, or character substring that XL FORTRAN associates with a variable, array element, or character substring in a blank common block
- A variable in a function subprogram whose name is the same as that of the function subprogram or an entry in the function subprogram
- A variable declared as **AUTOMATIC**.

You must not initialize a variable, array element, or character substring more than once in an executable program. If you associate two or more variables, array elements, or character substrings, you can only initialize one in a **DATA** statement.

Implied–DO in a DATA Statement

You can use an implied–**DO** list in a **DATA** statement to initialize the elements of an array. Its form is:



do_object
is an array element name, character substring name, or implied–**DO** list.

variable_name
is the name of an integer variable called the implied–**DO** variable.

integer_expr1 (start)
integer_expr2 (stop)
integer_expr3 (step)
are each an integer constant expression; the expression can contain implied–**DO** variables of other implied–**DO** lists that have this implied–**DO** list within their ranges.

The range of an implied–**DO** list is the list *do_object_list*. XL FORTRAN establishes the iteration count and the values of the implied–**DO** variable from *integer_expr1*, *integer_expr2*, and *integer_expr3*, the same as for a DO statement, except that the iteration count must be positive. (See "Executing a DO Statement" on page 75.) When XL FORTRAN executes the implied–**DO** list, it specifies the items in the *do_object_list* once for each iteration of the implied–**DO** list, with the appropriate substitution of values for any occurrence of the implied–**DO** variable.

Each subscript expression in the *do_object_list* must be an integer constant expression, except that the expression can contain implied–**DO** variables of implied–**DO** lists that have the subscript expression within their ranges.

Examples of the DATA Statement

```
      INTEGER Z(100),EVEN_ODD(0:9)
      LOGICAL FIRST_TIME
      DATA    FIRST_TIME / .TRUE. /
      DATA    Z  / 100* 0 /
C Implied–DO list
      DATA  (EVEN_ODD(J),J=0,8,2) / 5 * 0 /
     +      ,(EVEN_ODD(J),J=1,9,2) / 5 * 1 /
C Nested example
      DIMENSION TDARR(3,4)
      DATA ((TDARR(I,J),J=1,4),I=1,3) /12 * 0/
```

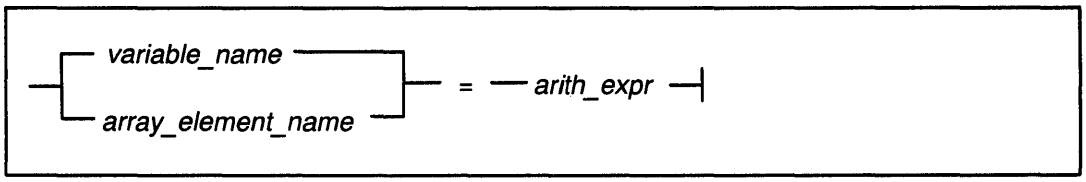
Chapter 8. Assignment Statements

Assignment statements are executable statements that assign values to variables, array elements, or character substrings.

This chapter describes the four kinds of assignment statements:

- Arithmetic
- Logical
- Character
- Statement label (ASSIGN).

Arithmetic Assignment



variable_name
array_element_name
is the name of a variable or array element of type integer, real, or complex.

arith_expr
is an arithmetic expression.

The arithmetic assignment statement causes evaluation of an arithmetic expression, conversion of the resulting value to the type of the variable or array element, and definition of the variable or array element by assigning that value to it, according to the rules in the following table.

Type of *b* / Type of *a*	**Integer**	**Real**	**Double Precision**	**Complex**	**Complex double precision**
Integer	Assign.	Fix and assign.	Fix and assign.	Fix and assign real part; imaginary part not used.	Fix and assign real part; imaginary part not used.
Real	Float and assign.	Assign.	Real assign.	Assign real part; imaginary part not used.	Real assign real part; imaginary part not used.
Double Precision	DP float and assign.	DP extend and assign.	Assign.	DP extend and assign to real part; imaginary part not used.	Assign real part; imaginary part not used.
Complex	Float and assign to real part; imaginary part set to 0.	Assign to real part; imaginary part set to 0.	Real assign real part; imaginary part set to 0.	Assign.	Real assign real and imaginary part.
Complex double precision	DP float and assign to real part; imaginary part set to 0.	DP extend and assign to real part; imaginary part not used.	Assign to real part; imaginary part set to 0.	DP extend and assign real and imaginary part.	Assign.

Figure 3. Conversion Rules for Arithmetic Assignment Statements (a=b)

The following defines the terms used in the figure:

a A variable or array element.

b An arithmetic expression.

Assign Transmit the expression value without change. If the expression value contains more significant digits than the variable a can hold, the value assigned to a is unpredictable.

Real assign Transmit to a as much precision of the most significant part of the expression value as REAL data can contain.

DP assign Transmit to a as much precision of the most significant part of the expression value as DOUBLE PRECISION data can contain.

Fix Truncate the fractional portion of the expression value and transform the result to an integer 4 bytes long. If the expression value contains more significant digits than a 4–byte integer can hold, the value assigned to the integer variable is unpredictable.

Float Transform the integer expression value to a REAL number, retaining in the process as much precision of the value as a REAL number can contain.

DP float Transform the integer expression value to a DOUBLE PRECISION number.

DP extend Extend the real value to a DOUBLE PRECISION number.

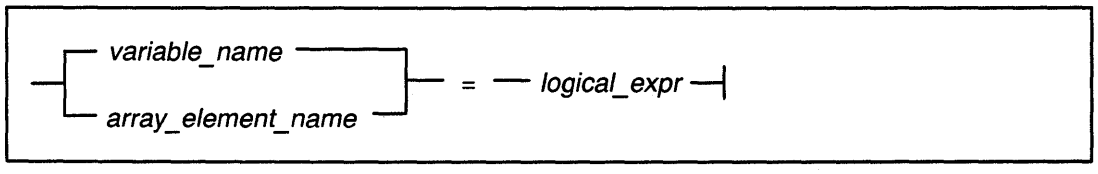
The following defines the data types used in the figure:

Integer	INTEGER INTEGER*1 INTEGER*2 INTEGER*4
Real	REAL REAL*4
Double Precision	DOUBLE PRECISION REAL*8 REAL*16
Complex	COMPLEX COMPLEX*8
Complex Double Precision	DOUBLE COMPLEX COMPLEX*16 COMPLEX*32

Examples of the Arithmetic Assignment Statement

```
ESQ = P**2 + M**2
ROOT(1) = (-B + SQRT(B**2 - 4.0*A*C))/(2.0*A)
```

Logical Assignment



variable_name
array_element_name
is the name of a variable or array element of type logical.

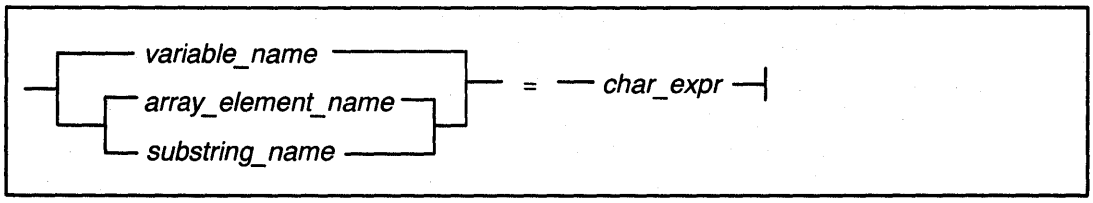
logical_expr
is a logical expression.

The logical assignment statement causes evaluation of a logical expression and definition of a variable or array element by assignment of the resulting value to the variable or array element. The value of the logical expression must be either true or false.

Example of the Logical Assignment Statement

```
LOGICAL INSIDE
.
.
REAL    R,RMIN,RMAX
INSIDE = (R .GE. RMIN) .AND. (R .LE. RMAX)
```

Character Assignment



variable_name, array_element_name, substring_name
is the name of a variable, array element, or substring of type character.

char_expr
is a character expression.

The character assignment statement causes evaluation of a character expression and definition of a character variable, character array element, or character substring by assignment of the resulting value to it.

You cannot reference any of the character positions you want defined in the character variable, character array element, or character substring by the character expression directly or through association of variables (using **COMMON**, **EQUIVALENCE**, or argument association).

If the length of the character variable, character array element, or character substring is greater than the length of the character expression, XL FORTRAN extends the character expression to the right with blanks until the lengths are equal, and then assigns them. If the length of the character variable, character array element, or character substring is less than the character expression, XL FORTRAN truncates the character expression on the right to match the length of the character variable, character array element, or character substring, and then assigns it.

You need only define as much of the character expression as is necessary to define the character variable, character array element, or character substring. For example:

```
CHARACTER SCOTT*4, DICK*8
SCOTT = DICK
```

This assignment of `DICK` to `SCOTT` requires that you define the substring `DICK(1:4)`. You do not have to define the rest of `DICK (DICK(5:8))`.

If you specify *substring_name*, XL FORTRAN assigns the character expression only to the character substring identified by that *substring_name*. The definition status of other character substrings does not change.

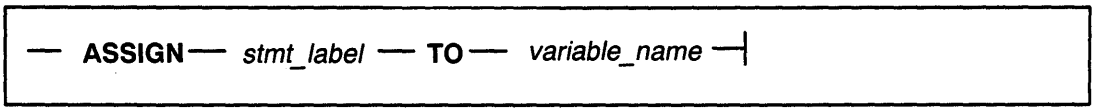
Examples of the Character Assignment Statement

```
CHARACTER*80 LINE, CH*1, SEQ*8
.
.
CH = LINE(1:1)
SEQ = LINE(73:80)
```

Typeless Constants in Assignment Statements

You can use typeless constants in arithmetic, logical, and character assignment statements. If the hexadecimal, octal, or binary constant is smaller than the length (in bytes) of the variable to be assigned, XL FORTRAN adds zeros on the left. If the constant is larger, the compiler truncates the hexadecimal, octal, or binary constant on the left. If the Hollerith constant is smaller than the length (in bytes) of the variable it is to be assigned to, XL FORTRAN adds blanks on the right. If the constant is larger, the compiler truncates the rightmost Hollerith characters.

Statement Label Assignment (ASSIGN)



stmt_label
specifies the statement label of an executable statement or a **FORMAT** statement in the program unit containing the **ASSIGN** statement.

variable_name
is the name of an **INTEGER*4** variable (not an array element).

The **ASSIGN** statement assigns a statement label to an integer variable. *stmt_label* must be a statement label of a statement appearing in the same program unit as the **ASSIGN** statement. If *stmt_label* is the statement label of an executable statement, you can use *variable_name* in an assigned **GO TO** statement. If *stmt_label* is the statement label of a **FORMAT** statement, you can use *variable_name* as the format identifier in a **READ**, **WRITE**, or **PRINT** statement with format control.

You can redefine an integer variable defined with a statement label value with the same or different statement label value or an integer value. However, you must define the variable with a statement label value when you reference it in an assigned **GO TO** statement or as a format identifier in an input/output statement.

The value of *variable_name* is not the integer constant represented by *stmt_label*, and you cannot use it as such. To use *variable_name* as an integer, you must first assign it an integer value using an arithmetic assignment statement or input statement. You can do this assignment directly, or through **EQUIVALENCE**, **COMMON**, or argument association.

Example of the Statement Label Assignment (ASSIGN) Statement

```
      ASSIGN 30 TO LABEL
      NUM = 40
      GO TO LABEL
      NUM = 50
30    ASSIGN 1000 TO IFMT
      WRITE(5,IFMT) NUM       !The value written out is 40.
1000  FORMAT(1X,I4)
```

Chapter 9. Control Statements

Control statements are executable statements that control the execution sequence of a program.

This chapter describes all the control statements except for **CALL** and **RETURN** (which are described in Chapter 10, "Program Units and Procedures"). The control statements described in this chapter are:

- Unconditional **GO TO**
- Computed **GO TO**
- Assigned **GO TO**
- Arithmetic **IF**
- Logical **IF**
- Block **IF**, **ELSE IF**, **ELSE**, and **END IF** (grouped in an **IF** construct)
- **DO**
- **DO WHILE**
- **END DO**
- **CONTINUE**
- **STOP**
- **PAUSE**
- **END**.

Unconditional GO TO

stmt_label
is the statement label of an executable statement in the same program unit as the unconditional **GO TO**.

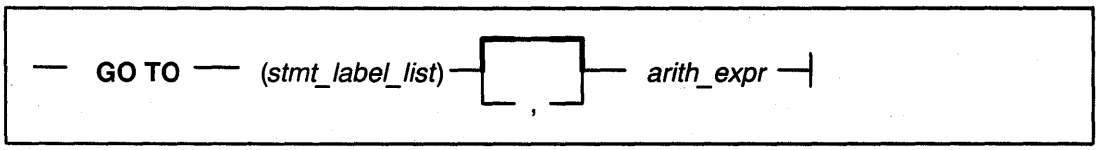
The unconditional **GO TO** statement transfers control to the statement identified by *stmt_label*. Every subsequent invocation of this **GO TO** statement results in a transfer to that same statement.

Any executable statement immediately following this statement must have a statement label; otherwise, you can never refer to or execute it.

The unconditional **GO TO** statement cannot be the terminal statement of a **DO** or **DO WHILE** loop.

Note: The **GO TO** statement can also appear as **GOTO**.

Computed GO TO



stmt_label
is the statement label of an executable statement in the same program unit as the computed **GO TO**. The same statement label can appear more than once in *stmt_label_list*.

arith_expr
is an arithmetic expression. If the value of the expression is noninteger, XL FORTRAN converts it to integer before using it.

Invocation of a computed **GO TO** statement causes evaluation of *arith_expr* and use of the value as an index into *stmt_label_list*. Control then transfers to the statement whose statement label you identify by the index. For example, if the value of *arith_expr* is 4, control transfers to the statement whose statement label is fourth in the *stmt_label_list*, provided there are at least 4 labels in the list.

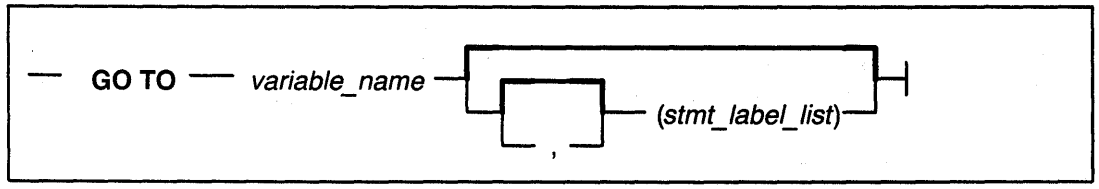
If the value of *arith_expr* is less than 1 or greater than the number of statement labels in the list, the **GO TO** statement has no effect (like a **CONTINUE** statement), and the next statement is executed.

The computed **GO TO** statement can be the terminal statement of a **DO** or **DO WHILE** loop.

Example of the Computed GO TO Statement

```
      INTEGER NEXT
      .
      .
      GO TO (100,200) NEXT
10    PRINT *,'Program transfers here if NEXT does not equal 1 or 2'
      .
      .
100   PRINT *,'Program transfers here if NEXT = 1'
      .
      .
200   PRINT *,'Program transfers here if NEXT = 2'
```

Assigned GO TO



variable_name
is a variable name of type **INTEGER*4** that you have assigned a statement label to in an **ASSIGN** statement.

stmt_label
is the statement label of an executable statement in the same program unit as the assigned **GO TO**. The same statement label can appear more than once in *stmt_label_list*.

At the time the assigned **GO TO** statement is executed, the variable you specify by *variable_name* with the value of a statement label must be defined. You must establish this definition with an **ASSIGN** statement in the same program unit as the assigned **GO TO** statement. The execution of the assigned **GO TO** statement transfers control to the statement identified by that statement label.

If *stmt_label_list* is present, the statement label assigned to the variable specified by *variable_name* must be one of the statement labels in the list. If, at the time of the invocation of the assigned **GO TO** statement, the current value of *variable_name* contains an integer value, assigned directly or through **EQUIVALENCE**, **COMMON**, or argument passing, the result of the **GO TO** is unpredictable. If the integer variable is a dummy argument in a subprogram, you must assign it a statement label in the subprogram, and also use it in an assigned **GO TO** in that subprogram.

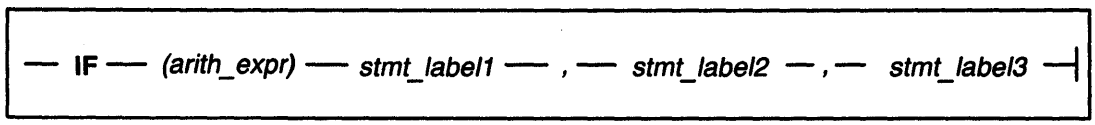
Any executable statement immediately following an assigned **GO TO** statement must have a statement label; otherwise, you can never refer to or execute it.

The assigned **GO TO** cannot be the terminal statement of a **DO** or **DO WHILE** loop.

Example of the Assigned GO TO Statement

```
      INTEGER RETURN_LABEL
      .
      .
c  Assign the return label and "call" the local procedure
      ASSIGN 100 TO RETURN_LABEL
      GOTO 9000
100   CONTINUE
      .
      .
9000  CONTINUE
c  A "local" procedure.
      .
      .
      GOTO RETURN_LABEL
```

Arithmetic IF



arith_expr
is an arithmetic expression of any type except complex.

stmt_label1
stmt_label2
stmt_label3
are statement labels of executable statements within the same program unit as the **IF** statement. The same statement label can appear more than once among the three statement labels.

The arithmetic **IF** statement causes evaluation of *arith_expr* and transfer of control to the statement identified by *stmt_label1*, *stmt_label2*, *or stmt_label3*, depending on whether the value of *arith_expr* is less than zero, zero, or greater than zero, respectively.

Any executable statement immediately following the arithmetic **IF** statement must have a statement label; otherwise, you can never refer to or execute it.

The arithmetic **IF** cannot be the terminal statement of a **DO** or **DO WHILE** loop.

Example of the Arithmetic IF Statement

```
      IF (K-100) 10,20,30
10    PRINT *,'K is less than 100.'
      GO TO 40
20    PRINT *,'K equals 100.'
      GO TO 40
30    PRINT *,'K is greater than 100.'
40    CONTINUE
```

Logical IF

logical_expr
is a logical expression.

stmt
is any unlabeled executable statement except **DO**, **DO WHILE**, **END DO**, block **IF**, **ELSE IF**, **ELSE**, **END IF**, **END**, or another logical **IF**.

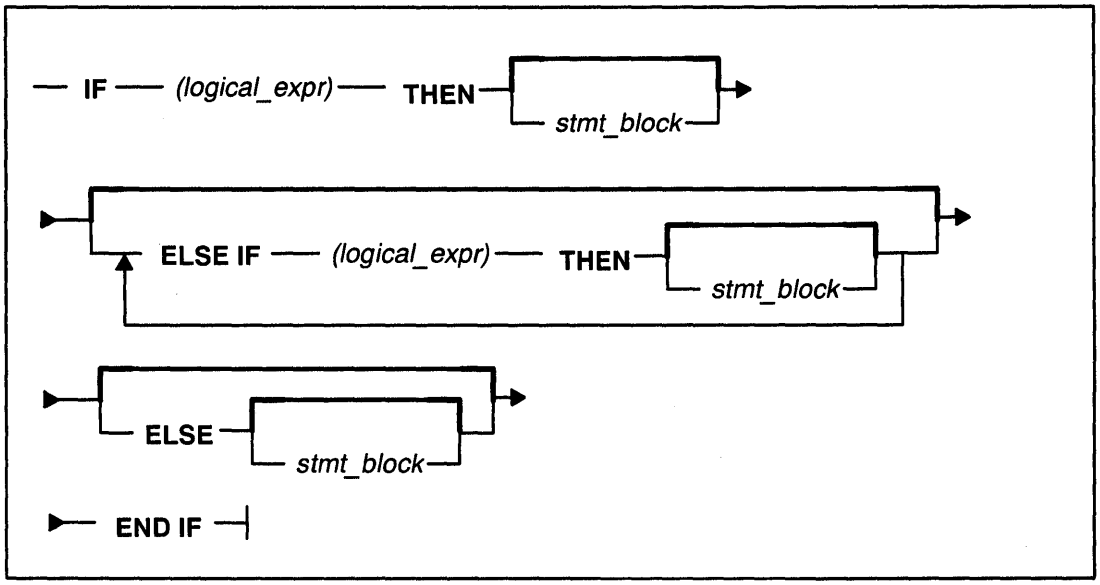
Execution of a logical **IF** statement causes evaluation of *logical_expr*. If the value of *logical_expr* is true, *stmt* executes. If the value of *logical_expr* is false, *stmt* does not execute and the **IF** statement has no effect (like a **CONTINUE** statement).

The statement *stmt* must not have a statement label, but the logical **IF** containing *stmt* can have a statement label. Execution of a function reference in *logical_expr* can change the values of variables, arrays, and array elements in *stmt*.

Example of the Logical IF Statement

```
      IF (ERR.NE.0) CALL ERROR(ERR)
```

IF Construct — Block IF, ELSE IF, ELSE, and END IF



logical_expr
is a logical expression.

stmt_block
is a statement block consisting of zero or more executable statements.

You use an **IF** construct to control the execution sequence. It is made up of a block **IF** statement, an **END IF** statement, and, optionally, **ELSE IF**, **ELSE**, and other executable statements. The box above shows the statements in an **IF** construct in their required sequence.

You can nest **IF** constructs; that is, any of the statement blocks can contain **IF** constructs.

The logical expressions in an **IF** construct are evaluated in the order of their appearance until it finds a true value, an **ELSE** statement, or an **END IF** statement:

- If it finds a true value or an **ELSE** statement, the statement block immediately following executes, and the **IF** construct is complete. The logical expressions in any remaining **ELSE IF** statements or **ELSE** statements of the **IF** construct are not evaluated. If the following statement block is empty, no statements execute, and the **IF** construct is complete.
- If it finds an **END IF** statement, no statement blocks execute, and the **IF** construct is complete.

You cannot transfer control into an **IF** construct from outside it. Transfer of control within an **IF** construct is permitted within statement blocks, but is not permitted between statement blocks or to an **ELSE IF** or **ELSE** statement.

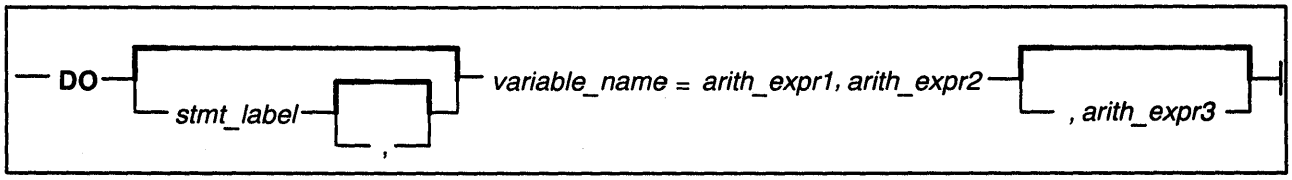
A block **IF**, **ELSE IF**, **ELSE**, or **END IF** cannot terminate a **DO** or **DO WHILE** loop. You cannot continue an **END IF** statement over more than one line so that the first line appears to be **END**.

Note: The **END IF** and **ELSE IF** statements can also appear as **ENDIF** and **ELSEIF** respectively.

Examples of the IF Construct

```
c Get a record (containing a command) from the terminal
100   CONTINUE
      .
      .
c Process the command
      IF (CMD .EQ. 'RETRY') THEN
           IF (LIMIT .GT. FIVE) THEN
c               Print retry limit exceeded
                .
                .
                CALL STOP
           ELSE
                CALL RETRY
           END IF
      ELSE IF (CMD .EQ. 'STOP') THEN
           CALL STOP
      ELSE IF (CMD .EQ. 'ABORT') THEN
           CALL ABORT
      ELSE
           GO TO 100
      END IF
```

DO



stmt_label
is the statement label of the terminal statement, which is the executable statement at the end of the **DO** loop.

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

arith_expr1 (start)
arith_expr2 (stop)
arith_expr3 (step)
are each an integer, real, or double precision expression.

The **DO** statement controls the execution of the statements that follow it, up to and including a specified terminal statement. These statements are called a **DO** loop.

The terminal statement must follow the **DO** statement and must be an executable statement. It cannot be any of the following statements: unconditional **GO TO**, assigned **GO TO**, arithmetic **IF**, block **IF**, **ELSE IF**, **ELSE**, **END IF**, **RETURN**, **STOP**, **END**, **DO**, or **DO WHILE**. If the terminal statement of a **DO** loop or **DO WHILE** loop is a logical **IF** statement, it can contain any executable statement except those statements to which the normal restrictions on the logical **IF** statement apply. (**DO**, **DO WHILE**, **END DO**, block **IF**, **ELSE IF**, **ELSE**, **ENDIF**, **END**, or another logical **IF**.)

You can use a labeled **END DO** statement as the terminal statement of a **DO** loop. If you omit the optional statement label *stmt_label*, the terminal statement of the **DO** loop must be an **END DO** statement.

More than one **DO** loop can share the same terminal statement if the terminal statement is not an **END DO** statement (labeled or unlabeled).

Range of a DO Loop

The range of a **DO** loop consists of all the executable statements following the **DO** statement, up to and including the terminal statement. Rules concerning the range of a **DO** loop are:

- If a **DO** statement appears within the range of a **DO** loop (that is, is nested), you must contain the range of the nested **DO** loop entirely within the range of the outer **DO** loop.
- If a **DO** statement appears within a statement block of an **IF** construct, you must contain the range of the **DO** loop entirely within that statement block.
- If an **IF** construct appears within the range of a **DO** loop, no part of the construct can appear outside the range.
- You cannot transfer control into the range of a **DO** loop from outside the range.
- You can only transfer control to a shared terminal statement from the innermost sharing **DO** loop.

Active and Inactive DO Loops

A **DO** loop is either active or inactive. Initially inactive, a **DO** loop becomes active only when its **DO** statement is executed. Once active, the **DO** loop becomes inactive only when:

- Its iteration count becomes zero.
- A **RETURN** statement occurs within the range of the **DO** loop.
- Control transfers to a statement in the same program unit but outside the range of the **DO** loop.
- A subroutine invoked from within the **DO** loop returns, through an alternate return specifier, to a statement that is outside the range of the **DO** loop.
- A **STOP** statement executes or the program stops for any other reason.

When a **DO** loop becomes inactive, the **DO** variable retains the last value assigned to it.

Executing a DO Statement

1. XL FORTRAN establishes the initial parameter, m_1, the terminal parameter, m_2, and the increment, m_3 by evaluating *arith_expr1*, *arith_expr2*, and *arith_expr3*, respectively. Evaluation includes, if necessary, conversion to the type of the **DO** variable according to the rules for arithmetic conversion in the figure on page 64. If you do not specify *arith_expr3*, m_3 has a value of 1. m_3 must not have a value of zero.
2. The **DO** variable becomes defined with the value of the initial parameter (m_1).
3. XL FORTRAN establishes the iteration count, and it is the value of the expression:

   $$\mathrm{MAX}\ (\mathrm{INT}\ (\ (m_2 - m_1 + m_3)\ /\ m_3),\ 0)$$

   Note that the iteration count is 0 whenever:

   $m_1 > m_2$ and $m_3 > 0$, or

   $m_1 < m_2$ and $m_3 < 0$

At the completion of the **DO** statement, loop control processing begins.

Loop Control Processing

Loop control processing determines if further execution of the range of the **DO** loop is required. The iteration count is tested. If the count is not zero, the first statement in the range of the **DO** loop begins execution. If the iteration count is zero, the **DO** loop becomes inactive. If, as a result, all of the **DO** loops sharing the terminal statement of this **DO** loop are inactive, normal execution continues with the execution of the next executable statement following the terminal statement. However, if some of the **DO** loops sharing the terminal statement are active, execution continues with incrementation processing.

Execution of the Range

Statements in the range of the **DO** loop are executed until the terminal statement is reached. Except by incrementation processing, you cannot redefine the **DO** variable nor can it become undefined during execution of the range of the **DO** loop.

Terminal Statement Execution

Execution of the terminal statement occurs as a result of the normal execution sequence, or as a result of transfer of control, subject to the restriction that you cannot transfer control into the range of a **DO** loop from outside the range. Unless execution of the terminal statement results in a transfer of control, execution then continues with incrementation processing.

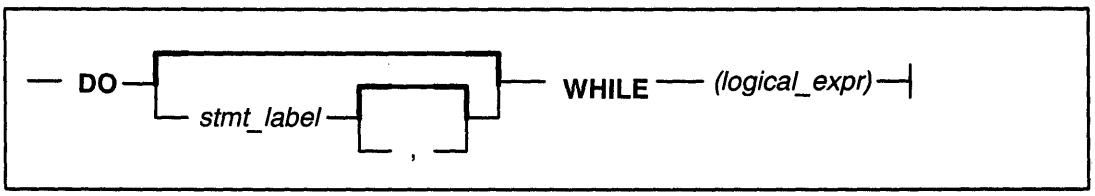
Incrementation Processing

1. The **DO** variable, the iteration count, and the increment (m_3) of the active **DO** loop whose **DO** statement was most recently executed, are selected for processing.
2. The value of the **DO** variable is incremented by the value of m_3.
3. The iteration count is decremented by 1.
4. Execution continues with loop control processing of the same **DO** loop whose iteration count was decremented.

Examples of the DO Statement

```
      DO I = 2, 5
         EARLIEST(I) = 0.0
         DO 10 J = 1, I-1
           IF (NETWORK(J,I) .NE. 0.0)
     x      EARLIEST(I) = MAX(NETWORK(J,I)+EARLIEST(J), EARLIEST(I))
10       CONTINUE
      END DO
```

DO WHILE



stmt_label
is the statement label of an executable statement appearing after the **DO WHILE** statement in the same program unit, and it denotes the end of the **DO WHILE** loop.

logical_expr
is any logical expression.

This statement indicates that you want the following range of statements, up to and including a specified terminal statement, to be repeatedly executed for as long as the logical expression specified in the statement continues to be true.

If you specify a statement label in the **DO WHILE** statement, you must terminate the **DO WHILE** loop with a statement that is labeled with that statement label. You can terminate a labeled **DO WHILE** statement with an **END DO** statement that is labeled with that statement label, but you cannot terminate it with an unlabeled **END DO** statement. If you do not specify a label in the **DO WHILE** statement, you must terminate the **DO WHILE** loop with an **END DO** statement.

The terminal statement of a **DO WHILE** loop must be an executable statement. The terminal statement of a **DO WHILE** loop must not be an unconditional **GO TO**, an assigned **GO TO**, arithmetic **IF**, block **IF**, **ELSE IF**, **ELSE**, **ENDIF**, **RETURN**, **STOP**, **END**, **DO** or **DO WHILE** statement. If the terminal statement of a **DO** loop is a logical **IF** statement, it can contain any executable statement except those statements to which the normal restrictions on the logical **IF** statement apply. (**DO**, **DO WHILE**, **END DO**, block **IF**, **ELSE IF**, **ELSE**, **ENDIF**, **END**, or another logical **IF**).

Nested **DO WHILE** loops can share the same terminal statement if the terminal statement is labeled, and it is not an **END DO** statement.

When the **DO WHILE** statement is executed, *logical_expr* is evaluated at the start of each iteration of the loop. If the value of the expression is true, the statements in the body of the loop are executed. If the value of the expression is false, control transfers to the statement following the loop.

Note: The **DO WHILE** statement can also appear as **DOWHILE**.

Examples of the DO WHILE Statement

```
      DO 10 WHILE (I .LE. 5)
         SUM = SUM + INC
         I = I + 1
10    END DO

      LOGICAL FLAG1
      DATA    FLAG1 /.TRUE./
      DO 20 WHILE (I .LE. 10)     ! Statement number is required
         X(I) = A
         I = I + 1
20    IF (.NOT. FLAG1) STOP
```

END DO

You can terminate a **DO** loop or a **DO WHILE** loop with the **END DO** statement.

If you label the **END DO** statement, you can use it as the terminal statement of a labeled or unlabeled **DO** or **DO WHILE** loop. An **END DO** statement terminates the innermost **DO** or **DO WHILE** loop only.

If a **DO** or **DO WHILE** statement does not specify a statement label, the terminal statement of the **DO** or **DO WHILE** loop must be an **END DO** statement. The labeled or unlabeled **END DO** statement terminates the innermost **DO** loop only.

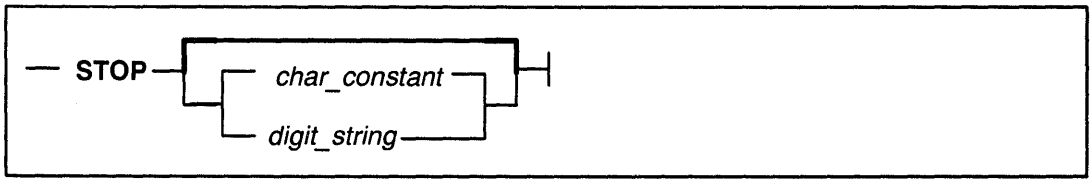
You must not continue an **END DO** statement over more than one line in such a manner that the initial line appears to be an **END** statement.

Note: The **END DO** statement can also appear as **ENDDO**.

CONTINUE

This statement is an executable control statement that takes no action. You can use it to designate the end of a **DO** loop or **DO WHILE** loop, or to label a position in a program. Execution of a **CONTINUE** statement has no effect.

STOP



char_constant
is a character constant enclosed in apostrophes or double quotation marks, and containing alphanumeric and/or special characters.

digit_string
is a string of one through five digits.

The **STOP** statement causes the program to stop execution and display *char_constant* or *digit_string*, if specified. A **STOP** statement cannot terminate the range of a **DO** loop or **DO WHILE** loop.

When the **STOP** statement `STOP 'here.'` is processed, the following appears on the terminal screen:

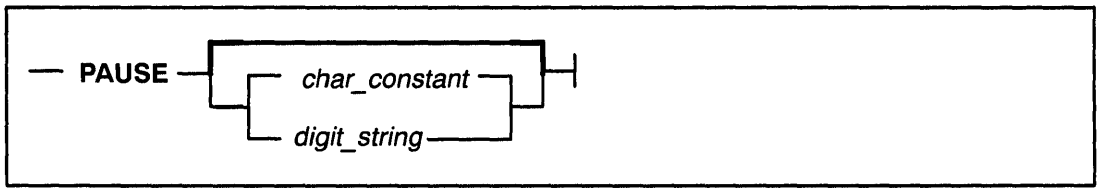
```
STOP here.
```

If you specify *digit_string,* XL FORTRAN sets the return code to **MOD**(*digit_string*,256) because the AIX return code is only 1 byte.

Examples of the STOP Statement

```
STOP 'Abnormal Termination'
STOP 15
```

PAUSE



char_constant
is a character constant enclosed in apostrophes or double quotation marks, and containing alphanumeric and/or special characters.

digit_string
is a string of one through five digits.

The **PAUSE** statement temporarily suspends the execution of a program and displays *char_constant* or *digit_string*, if you specify either.

When the **PAUSE** statement `PAUSE 'here.'` is processed, the following appears on the terminal screen:

```
PAUSE here.
```

Processing continues when you press the **ENTER** key.

Examples of the PAUSE Statement

```
PAUSE 'Insert a diskette into the default drive.'
PAUSE 10
```

END

The **END** statement is the final statement in a program unit. It is the only required statement in a program unit.

The **END** statement is an executable statement, and you can label it. You cannot continue an **END** statement, and no other statement in a program unit can have an initial line that appears to be an **END** statement.

An **END** statement in a main program terminates the execution of the program. An **END** statement in a function or subroutine subprogram has the same effect as a **RETURN** statement. (See "RETURN Statement" on page 90 for further information.) The last line of every program unit must be an **END** statement. An inline comment, initiated by `!`, can appear on the same line as an **END** statement. Any comment line appearing after an **END** statement belongs to the next program unit.

Example of the END statement with an end–of–line comment

```
END                    ! the END of the program unit
```

Chapter 10. Program Units and Procedures

This chapter describes:

- Relationships among program units and procedures
- Functions and subroutines
- Arguments
- Recursion
- The **PROGRAM**, **FUNCTION**, statement function, **SUBROUTINE**, **CALL**, **ENTRY**, **RETURN**, and **BLOCK DATA** statements.

Relationships Among Program Units and Procedures

A program unit is a sequence of statements and optional comment lines, with the final statement an END statement. An executable program is a collection of program units consisting of one main program and zero or more subprograms.

Program unit relationships are illustrated in Figure 4 below.

A procedure can be invoked by a program unit to perform a particular activity. When a procedure reference is made, the referenced procedure is executed.

Procedure relationships are illustrated in Figure 5 on page 82.

XL FORTRAN allows recursion if you specify the **RECUR** compiler option. (See "Recursion" on page 95.)

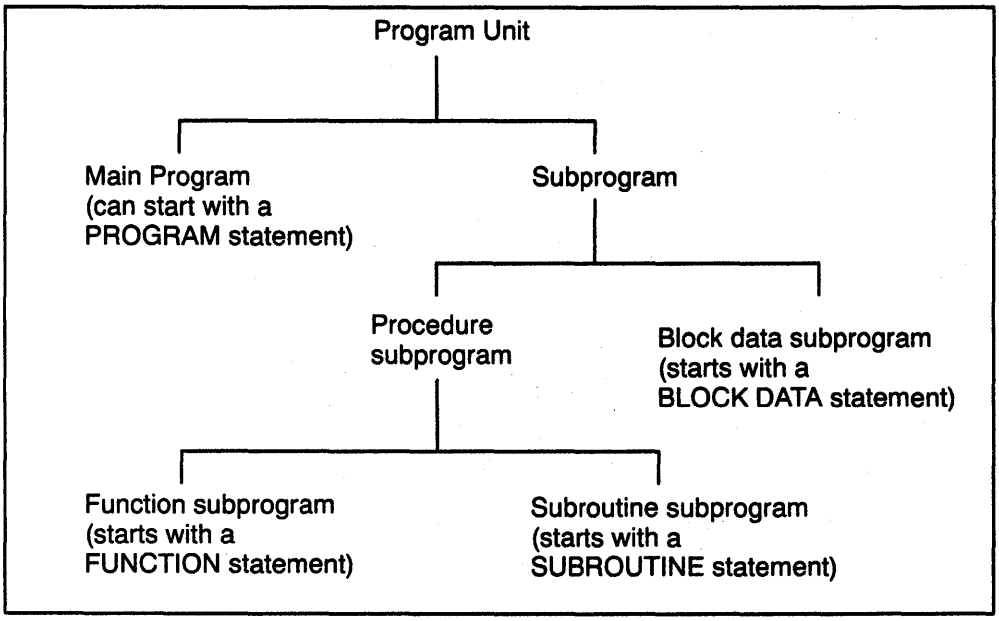


Figure 4. Program Unit Relationships

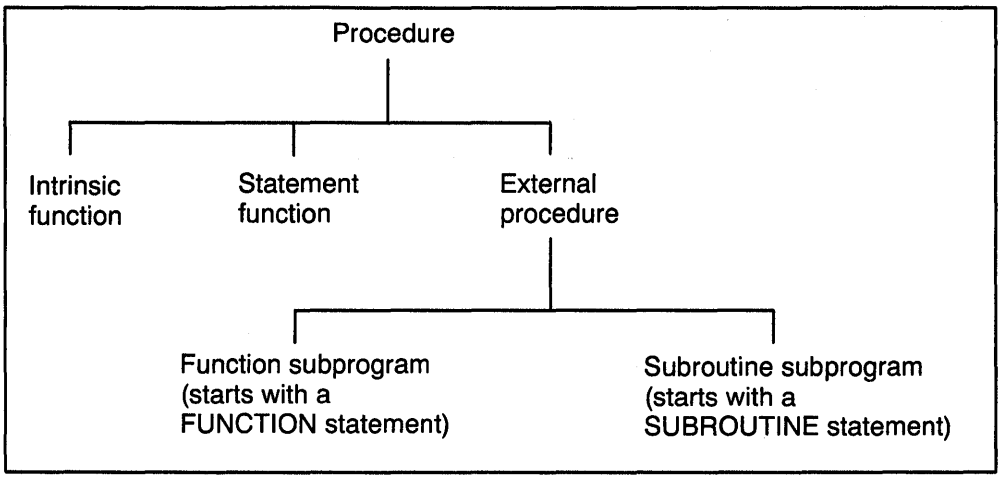


Figure 5. Procedure Relationships

PROGRAM Statement — Main Program

name
is the name of the main program in which this statement appears.

A main program is the program unit that receives control from the system when the executable program is invoked at run time. A main program can contain any statement except **BLOCK DATA**, **FUNCTION**, **SUBROUTINE**, or **ENTRY**. A **RETURN** statement can appear in a main program. It has the same effect as a **STOP** statement. The appearance of a **SAVE** statement has no effect.

The **PROGRAM** statement specifies that a program unit is a main program. The **PROGRAM** statement is optional; if specified, it must be the first statement of the main program.

If a **PROGRAM** statement is not specified, the name of the main program is `MAIN`. If the name is specified, it cannot be the same as any other name in this program. The name, if specified, has no type and is not affected by any specification statements. You cannot refer to a main program from a subprogram or from itself.

Example of the PROGRAM Statement

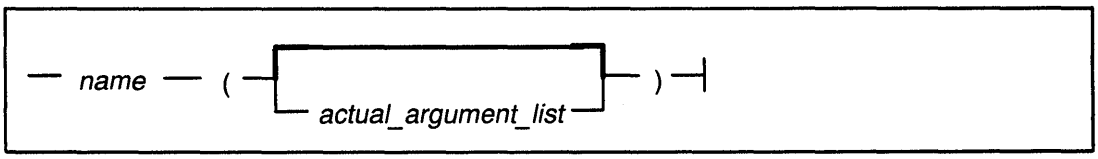
```
PROGRAM SCALE
```

Functions

A function is a procedure that is invoked by its name in a function reference. It then returns a value to the point of reference. The three kinds of functions are intrinsic functions (see Appendix A, "Intrinsic Functions"), statement functions, and external functions (or function subprograms).

Function Reference

A function reference is used as a primary in an expression to invoke a function. The form of a function reference is:



name
is the name of an external function, an entry in an external function, a statement function, or an intrinsic function.

actual_argument
is an actual argument, described on page 91.

Executing a function reference results in the following order of events:

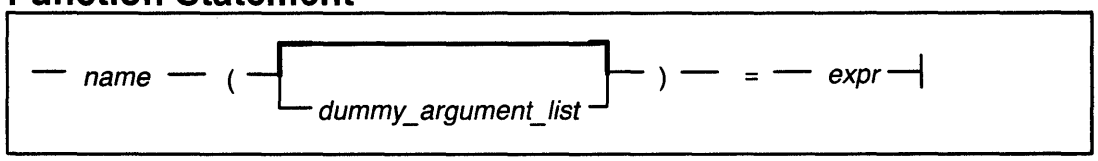
1. Actual arguments that are expressions are evaluated.
2. Actual arguments are associated with their corresponding dummy arguments.
3. The referenced function is executed.
4. The value of the function (called the function value) is available to the referencing expression.

Execution of a function reference must not alter the value of any other data item within the statement in which the function reference appears. However, invocation of a function reference in the expression of a logical **IF** statement can affect variables, arrays, and array elements in the statement that is executed when the value of the expression is true.

The argument list keywords **%VAL** and **%REF** are supplied to aid the interlanguage calls by allowing arguments to be passed by value and reference respectively. (See "%VAL and %REF" on page 93.)

See "Examples of Statement Function Statements" on page 84 and "Examples of the FUNCTION Statement" on page 86 for examples of function references.

Statement Function Statement



name
is the name of the statement function.

dummy_argument
is a statement function dummy argument. See page 91 for a description of dummy arguments.

expr
is any arithmetic, logical, or character expression.

A statement function is a single–statement function that is internal to the program unit in which it is defined. It is defined by a statement function statement and invoked by a function reference. All statement function definitions must follow the specification statements and precede the first executable statement of the program unit.

name determines the data type of the value returned from the statement function. If the data type of *name* does not match that of *expr*, the value of *expr* is converted to the type of *name* in accordance with the rules for assignment statements. See Chapter 8, "Assignment Statements". If the **IMPLICIT NONE** statement is used within the program unit, the type of the statement function must be explicitly declared in an explicit type statement.

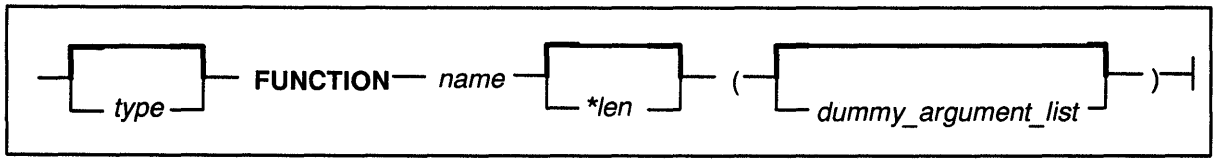
An external function reference in *expr* must not cause the actual argument of the statement function to become undefined or redefined. The expression *expr* cannot contain a reference to the statement function being defined. If the statement function definition is in a subprogram, *expr* cannot contain a reference to that subprogram's name or the name of any of its entry points.

The name of a statement function of type character must not have a length specifier of an asterisk in parentheses. You cannot use *name* to identify any other entity in the program unit except a common block, and it must not appear in any specification statement except an explicit type statement. You cannot use *name* as an actual argument.

Examples of Statement Function Statements

```
      PARAMETER (PI = 3.14159)
      REAL AREA,CIRCUM,R,RADIUS
C Define statement functions AREA and CIRCUM.
      AREA(R) = PI * (R**2)
      CIRCUM(R) = 2 * PI * R
      .
      .
C Reference the statement functions.
      PRINT *,'The area is: ',AREA(RADIUS)
      PRINT *,'The circumference is: ',CIRCUM(RADIUS)
```

FUNCTION Statement — Function Subprogram (External Function)



type

specifies the data type of the value that the function subprogram returns. *type* can be **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **DOUBLE COMPLEX**, **LOGICAL**, or **CHARACTER**[**char_len*], where *char_len* is the length specification of the result of the character function. *char_len* can have any of the forms allowed in a **CHARACTER** type statement (see page 50), except that an integer constant expression must not include the name of a constant. The default for *char_len* is 1. In the calling program, the length associated with the function name must be the same as *char_len*. If a **CHARACTER** function has a length that is an asterisk, the name cannot be used in a concatenation except in a character assignment statement. If the name is of type character, all entry names must be of type character, and the lengths must be the same. If one length is specified as an asterisk, all lengths must be specified as asterisks.

name

is the name of the function subprogram. *name* can appear in a type statement, but in no other nonexecutable statement.

len

is a positive, nonzero, unsigned integer constant. It represents the permissible length specifications for its associated type. It can be included only when the type is specified. It must not be used when **DOUBLE PRECISION**, or **DOUBLE COMPLEX** is specified. * *len* overrides the existing specified length.

dummy_argument

is a dummy argument, described on page 91.

A function subprogram, or external function, is a program unit that specifies a function. A function subprogram is invoked by a function reference and returns a value to the invoking program unit. For the purpose of returning the function value, the function name and any entry names are considered to be variable names, and a value must be assigned to one of those names during every invocation of the function. Also, a value can be assigned to one or more of its dummy arguments for the purpose of returning values. The function or any of its dummy arguments can be assigned values in any of the following ways:

- As the variable name to the left of the equal sign in an arithmetic, logical, or character assignment statement
- As the argument of a **CALL** statement or function reference that will cause a value to be assigned in the subprogram referenced
- In the list of a **READ** statement within the subprogram
- As one of the parameters in an **INQUIRE** statement that is assigned a value within the subprogram
- As a **DO** variable or an implied–**DO** variable in an input/output statement
- As the result of the **IOSTAT** specification in an input/output statement
- As the result of the **NUM** specification in an input/output statement.

The first statement of a function subprogram must be a **FUNCTION** statement. A function subprogram can contain any statement except another **FUNCTION**, **PROGRAM**, **SUBROUTINE**, or **BLOCK DATA** statement.

The name of a function subprogram determines the data type of the value returned from the subprogram. If the **IMPLICIT NONE** statement is used within the function subprogram, the type of the function must be explicitly declared in either the **FUNCTION** statement or an explicit type statement. The name of a function must not appear in any other nonexecutable statement except an explicit type statement.

The variable whose name is the name of the function is associated with any variables whose names are also entry names. This is called entry association. The definition of any one of them becomes the definition of all the associated variables having that same type, and is the value of the function no matter at which entry point it was entered. Such variables are not required to be of the same type unless the type is character, but the variable whose name is used to reference the function must be in a defined state when a **RETURN** or **END** statement is executed in the subprogram. An associated variable of a different type must not become defined during the execution of the function reference, unless an associated variable of the same type redefines it later during the execution of the subprogram.

Because you can have recursion when the **RECUR** compiler option is specified, the function name can be referenced within the function subprogram in a context that requires the function to be evaluated. If the function name is used as an actual argument, that argument is a variable and not a procedure name being passed to the subprogram.

Examples of the FUNCTION Statement

```
CHARACTER*10 FUNCTION SUFFIX(STR)
CHARACTER*7  STR
SUFFIX = STR // 'SUF'
END
FUNCTION VALID1(ARG1)
COMMON /ARG1/ A           !dummy argument can be the same as a
VALID1 = A                !        common block name
END
FUNCTION FNAME(DUMMY)
FNAME = 1.0               !not a recursive reference, but a
DUMMY = FNAME             !      scalar reference
END
REAL FUNCTION CUBE*16()
COMMON /COM1/ A
CUBE = A * A * A
END
FUNCTION FNAME(DUMMY)
 . . .                                    !recursive reference
IF (DUMMY.LT.10) DUMMY = FNAME(DUMMY+1)
FUNCTION RIGHT1(ARG1)
 . . .                                 !argument is a variable,
CALL SUB(RIGHT1)                       !   not a procedure
```

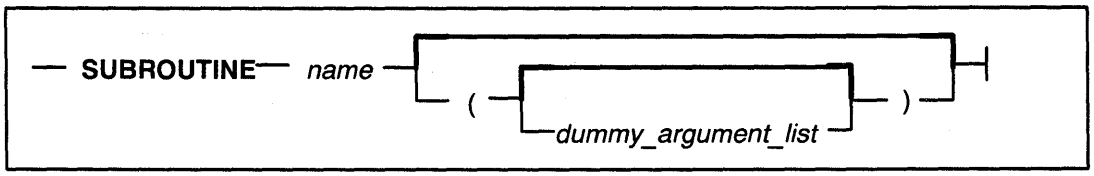
Main Program

```
      PROGRAM MAIN
c Actual args are X2, X1, X0
     REAL ROOT,X2,X1,X0
      .
      .
c     2*(X**2) + 4.5*X + 1
      X2 = 2.0
      X1 = 4.5
      X0 = 1.0
 c Reference function sub.
      ROOT = QUAD(X2,X1,X0)
      .
      .
```

Function Subprogram

```
c Dummy args are A, B, and C
     REAL FUNCTION QUAD(A,B,C)
      REAL A,B,C
     QUAD = (-B + SQRT(B**2-4*A*C)) / (2*A)
      RETURN
      END
```

SUBROUTINE Statement



name
is the name of the subroutine subprogram.

dummy_argument
is a dummy argument, described on page 91.

A subroutine subprogram, or subroutine, is a program unit that is invoked by its name or one of its entry names in a **CALL** statement.

The first statement of a subroutine subprogram must be a **SUBROUTINE** statement. A subroutine subprogram can contain any statement except **PROGRAM**, **FUNCTION**, **BLOCK DATA**, or another **SUBROUTINE** statement. The subroutine name cannot appear in any other statement in the subroutine subprogram, unless recursion has been specified. It must not be the same as any name in the program unit, the **PROGRAM** name, a subprogram name, or common block name in any other program unit of the executable program.

If the **RECUR** compiler option has been specified, the subroutine name can appear in a **CALL** statement within the subroutine subprogram. The subroutine name cannot appear as an actual argument in a **CALL** statement or function reference, because it cannot appear in an **EXTERNAL** statement. (See "Recursion" on page 95.)

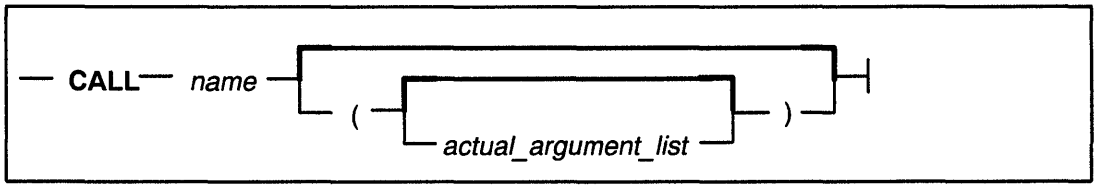
A subroutine subprogram can use one or more of its dummy arguments to return values to the invoking program unit. A value can be assigned to one of the dummy arguments during invocation of the subroutine subprogram in one of the following ways:

- As the variable name to the left of the equal sign in an arithmetic, logical, or character assignment statement
- As the argument of a **CALL** statement or function reference that will cause a value to be assigned in the subprogram referenced
- In the list of a **READ** statement within the subprogram
- As one of the parameters in an **INQUIRE** statement that is assigned a value within the subprogram
- As a **DO** variable or an implied–**DO** variable in an input/output statement
- As the result of the **IOSTAT** specification in an input/output statement
- As the result of the **NUM** specification in an input/output statement.

Example of the SUBROUTINE Statement

```
SUBROUTINE FIT(J,E,B)
```

CALL Statement



name
is the name of a subroutine or an entry in a subroutine subprogram. This name can be a dummy argument in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement.

actual_argument
is an actual argument, described on page 91.

Executing a CALL statement results in the following order of events:

1. Actual arguments that are expressions are evaluated.
2. Actual arguments are associated with their corresponding dummy arguments.
3. Control transfers to the specified subroutine.
4. The subroutine is executed.
5. Control returns from the subroutine.

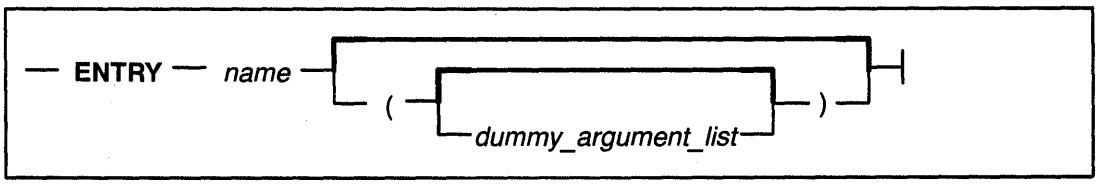
A main program cannot call itself or be called by a subprogram, but a subprogram can refer to itself directly if the **RECUR** compiler option is specified. (See "Recursion" on page 95.)

The argument list keywords **%VAL** and **%REF** are supplied to aid interlanguage calls by allowing arguments to be passed by value and reference respectively. (See "%VAL and %REF" on page 93.)

Examples of the CALL Statement

```
CALL SUB1                          !subroutine call with no arguments
CALL SUB1()                        !
                                   !
CALL SUB4(.FALSE.)                 !subroutine call with logical
CALL SUB4(I.EQ.J)                  !         argument
                                   !
CALL SUB5(C1//C2)                  !subroutine call with character
                                   !         argument
                                   !
CALL SUB8(I,J,*100,*200,*300)      !subroutine call with alternate
                                   !         returns
                                   !
EXTERNAL FUNC                      !procedure name can be passed by
CALL RIGHT2(%REF(FUNC))            !         reference
                                   !
COMPLEX XVAR                       !complex argument can be passed
CALL RIGHT3(%REF(XVAR))            !         by reference
                                   !
IVARB=6                            !integer argument can be passed
CALL TPROG(%VAL(IVARB))            !         by value
```

ENTRY Statement



name
is the name of an entry point in a function subprogram or subroutine subprogram. It is called an entry name.

dummy_argument
is a dummy argument, described on page 91.

A function subprogram or subroutine subprogram has a primary entry point that is established through the **SUBROUTINE** or **FUNCTION** statement. The **ENTRY** statement establishes an alternate entry point. Therefore, the **ENTRY** statement cannot appear in a main program or **BLOCK DATA** subprogram.

An **ENTRY** statement can appear anywhere after the **FUNCTION** or **SUBROUTINE** statement, except in a **DO** loop range, a **DO WHILE** loop, or within a block **IF** structure. **ENTRY** statements are nonexecutable and do not affect control sequencing during the execution of a subprogram. In a function subprogram, the entry name must not appear in any statement preceding the **ENTRY** statement except in a type statement. **ENTRY** statements can appear before the **IMPLICIT** or **PARAMETER** statements. The appearance

of an **ENTRY** statement does not alter the rule that statement functions must precede the first executable statement.

In a function subprogram, *name* identifies an external function and can be referenced (invoked) as an external function from the calling program unit. In a subroutine subprogram, *name* identifies a subroutine and can be referenced as a subroutine from the calling program unit. When the reference is made, execution begins with the first executable statement following the **ENTRY** statement.

If the **RECUR** compiler option is specified, a subprogram can reference any of its **ENTRY** names. If the compiler option is not specified, a subprogram must not refer to itself, or any of its entry points, either directly. (See "Recursion" on page 95.)

The name of an entry in a function subprogram must appear as a variable name in the function subprogram and you must define it upon exit from the subprogram, when the function is invoked through that entry. At least one name of the set of all **FUNCTION** and **ENTRY** names must be assigned a value. The name of an entry cannot be a dummy argument or be in an **EXTERNAL** statement. The name of an entry cannot appear in a common block, statement function, **DATA** statement, or **SAVE** statement.

In a function subprogram, *name* can be typed by an explicit type statement or an **IMPLICIT** statement. If **IMPLICIT NONE** has been specified in a function subprogram, the **FUNCTION** name and any **ENTRY** names must be explicitly typed.

If an entry name in a function subprogram is of type character, all entry names in the subprogram and the name of the subprogram must be of type character. If the length specifier of an entry named in the function subprogram or the name of the subprogram itself is an asterisk in parentheses (indicating inherited length), all entry names and the subprogram name must have a length specifier of an asterisk in parentheses. Otherwise, all such names must have a length specification of the same integer value.

In a function subprogram of noncharacter type, the entry names, and the function name can be of different types, and the names are treated as if they appeared in an **EQUIVALENCE** statement. **ENTRY** names within subroutine subprograms cannot be explicitly typed.

Entry into a subprogram associates actual arguments with the dummy arguments of the referenced **ENTRY** statement. Therefore, all appearances of these arguments in the subprogram become associated with actual arguments. You can use argument list keywords ("%VAL and %REF" on page 93) as actual arguments.

A name in the *dummy_argument_list* must not also appear:

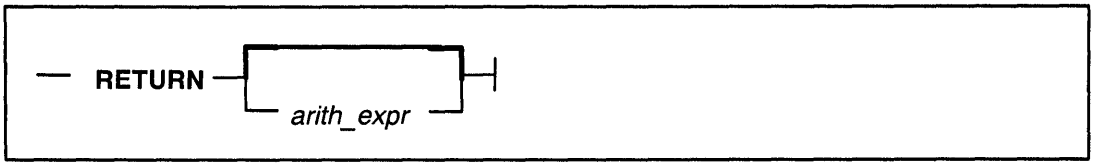
- In an executable statement preceding the **ENTRY** statement unless it also appears in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement that precedes the executable statement.
- In the expression of a statement function statement unless the name is also a dummy argument of the statement function, appears in a **FUNCTION** or **SUBROUTINE** statement, or appears in an **ENTRY** statement that precedes the statement function statement.

The number of dummy arguments and their data types in the *dummy_argument_list* of this **ENTRY** statement, or other **ENTRY** statements, and of the primary entry point, can differ. The actual arguments for each **CALL** or function reference must agree in order, type, and number with the dummy arguments in the **SUBROUTINE**, **FUNCTION**, or **ENTRY** statement to which it refers. All dummy arguments of the same name in a subprogram, regardless of their being in an **ENTRY**, **SUBROUTINE**, or **FUNCTION** statement, refer to the same dummy argument.

Example of the ENTRY Statement

```
REAL FUNCTION VOL(RDS,HGT)
PARAMETER (PI = 3.14159)
REAL RDS,HGT
A(RDS) = PI * RDS**2
VOL= A(RDS) * HGT
RETURN
ENTRY AREA(RDS)
AREA = A(RDS)
RETURN
END
```

RETURN Statement



arith_expr
is any arithmetic expression. If the value of the expression is noninteger, it is converted to integer before use.

The **RETURN** statement:

- In a function subprogram, it ends the execution of the subprogram and transfers control back to the referencing statement. The value of the function is available to the referencing program unit.
- In a subroutine subprogram, it ends the subprogram and transfers control to the first executable statement after the **CALL** statement or to an alternate return point, if one is specified. (See "Arguments" on page 91 for a description of alternate return points.)

arith_expr can be specified in a subroutine subprogram only and it specifies an alternate return point. Letting *m* be the value of *arith_expr*, if $1 \le m \le$ the number of asterisks in the **SUBROUTINE** or **ENTRY** statement, the *m*th asterisk in the dummy argument list is selected. Control then returns to the invoking program unit at the statement whose statement label is specified as the *m*th alternate return specifier in the **CALL** statement. For example, if the value of *m* is 5, the fifth asterisk in the dummy argument list is selected, and control returns to the statement whose statement label is specified as the fifth alternate return specifier in the **CALL** statement.

If *arith_expr* is omitted or if its value (*m*) is not in the range 1 through the number of asterisks in the **SUBROUTINE** or **ENTRY** statement, a normal return is executed. Control returns to the invoking program unit at the statement following the **CALL** statement.

Executing a **RETURN** statement terminates the association between the dummy arguments of the subprogram and the current actual arguments. All entities within the subprogram become undefined except:

- Entities given an initial value in a **DATA** or explicit specification statement and where initial values were not changed
- Entities in blank common
- Entities in named common that appear in the subprogram and appear in at least one other program unit that is referring either directly or indirectly to the subprogram

- Entities specified as **STATIC** (Recall that all variables are static by default.)
- Entities specified in a **SAVE** statement.

A subprogram can contain more than one **RETURN** statement, but it does not require one. An **END** statement in a function or subroutine subprogram has the same effect as a **RETURN** statement. A **RETURN** statement in a main program is equivalent to the execution of a **STOP** statement.

Arguments

An actual argument appears in the argument list of a procedure reference. An actual argument in an external function reference can be one of the following:

- An arithmetic, logical, or character expression, excluding a character expression involving concatenation of an operand whose length specifier is an asterisk in parentheses (indicating inherited length)
- A variable, array element, array name, or constant
- An intrinsic function name except for those listed under "INTRINSIC" on page 55
- An external procedure name
- A dummy procedure name
- If the actual argument is in a **CALL** statement, an alternate return specifier, having the form **stmt_label*, where *stmt_label* is the statement label of an executable statement in the same program unit as the **CALL** statement
- An argument list keyword. (See "%VAL and %REF" on page 93.)

An actual argument in a statement function or intrinsic function reference can be one of the following:

- An arithmetic, logical, or character expression, excluding a character expression involving concatenation of an operand whose length specifier is an asterisk in parentheses (indicating inherited length)
- A variable, array element, or constant.

You can also use a "typeless" constant as an actual argument. If the typeless constant is a hexadecimal, octal, or binary constant, and is smaller than 4 bytes in length, XL FORTRAN adds zeros on the left. If the constant is larger than 4 bytes, the compiler truncates the leftmost hexadecimal, octal, or binary digits.

If a Hollerith constant is passed by value, and is smaller than 4 bytes in length, XL FORTRAN adds blanks on the right. If the constant is larger than 4 bytes, and passed by value, the compiler truncates the rightmost Hollerith characters. The default for passing Hollerith constants is as if they are character actual arguments.

A dummy argument appears in the argument list of a procedure. A dummy argument is specified in a statement function statement, **FUNCTION** statement, **SUBROUTINE** statement, or **ENTRY** statement. Dummy arguments in statement functions, function subprograms, and subroutine subprograms indicate the types of actual arguments and whether each argument is a single value, array of values, procedure, or statement label. A dummy argument in an external function definition is classified as one of the following:

- A variable name
- An array name
- A procedure name
- An asterisk (in subroutines only, to indicate an alternate return point).

A dummy argument in a statement function definition is classified as the following:

- A variable name.

A given name can appear only once in a dummy argument list.

A statement function dummy argument name must not be used in any specification statement except an explicit type statement. A dummy argument name must not be the same as the procedure name appearing in a **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statement in the same program unit. The scope of a statement function dummy argument is the scope of the statement function statement.

A dummy argument name that appears in a **FUNCTION** or **SUBROUTINE** statement cannot appear in an **EQUIVALENCE**, **COMMON**, **DATA**, **PARAMETER**, **SAVE**, **INTRINSIC**, or **NAMELIST** statement, except as a common block name.

A character dummy argument of inherited length must not be used as an operand for concatenation, except in a character assignment statement.

See "Examples of the FUNCTION Statement" on page 86 for an example of arguments.

Association of External Procedure Arguments

Actual arguments are associated with dummy arguments when a function or subroutine is referenced (invoked). The first actual argument becomes associated with the first dummy argument, the second actual argument with the second dummy argument, and so forth. Argument association within a program unit terminates at the invocation of a **RETURN** or **END** statement in the program unit. There is no retention of argument association between one reference of a subprogram and the next reference of the subprogram. The subprogram reserves no storage for the dummy argument. It uses the corresponding actual argument for calculations. Therefore, the value of the actual argument changes when the dummy argument changes.

Actual arguments must agree in number, order, and type with their corresponding dummy arguments, except for two cases: a subroutine name has no type and must be associated with a dummy procedure name, and an alternate return specifier has no type and must be associated with an asterisk.

Argument association can be carried through more than one level of procedure reference.

If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument can become defined during that subprogram. For example, if a subroutine definition is:

```
SUBROUTINE XYZ (A,B)
```

and it is referenced by:

```
CALL XYZ (C,C)
```

then the dummy arguments `A` and `B` each become associated with the same actual argument `C` and, therefore, with each other. Neither `A` nor `B` can become defined during the execution of subroutine `XYZ` or by any procedures referenced by `XYZ`.

If a subprogram reference causes a dummy argument to become associated with a data item in a common block in the referenced subprogram or in a subprogram referenced by the referenced subprogram, neither the dummy argument nor the data item in the common block can become defined within the subprogram or within a subprogram referenced by the referenced subprogram.

%VAL and %REF

To call subprograms written in languages other than FORTRAN (for example, user–written C programs, or AIX system routines), the actual arguments may need to be passed by a method different from the default method used by FORTRAN. The form of an actual argument can be changed by using the **%VAL** and **%REF** keywords in the argument list of a **CALL** statement or function reference. These keywords specify the way the actual argument should be passed to the subprogram.

The argument list keywords are:

%VAL This keyword causes the actual argument to be passed as 32–bit intermediate values. If the actual argument is an integer that is shorter than 32 bits, it is sign–extended to a 32–bit value. If the actual argument is a logical that is shorter than 32 bits, it is padded on the left with zeros to a 32–bit value. If the actual argument is of type real or complex with a length greater than 32 bits, it is passed as multiple 32–bit intermediate values.

This keyword can be used with actual arguments that are **CHARACTER*1**, logical, integer, real, or complex expressions. It cannot be used with actual arguments that are array names, procedure names, or character expressions of length greater than 1 byte. Hexadecimal, binary, and octal constants are passed as if they were **INTEGER*4**. If the actual argument is a **CHARACTER*1**, it is padded on the left with zeros to a 32–bit value.

%REF This keyword causes the actual argument to be passed by reference. (The address of the actual argument is passed.) This is the default for FORTRAN.

Note that, if the actual argument is of character data type, only the address of the actual argument is passed, whereas a character actual argument passed without the **%REF** function is passed as the address and the length of the character argument. If such a character argument is being passed to a C routine, the string must be terminated with a null character, so that the C routine can determine the length of the string.

Length of Character Arguments

If arguments are of type character, the lengths of the actual arguments must be greater than or equal to the lengths of the dummy arguments. If an actual argument is longer, only the leftmost characters are associated with the dummy argument.

If a dummy argument has a length specifier of an asterisk in parentheses, the length of the dummy argument is "inherited" from the actual argument. The length is inherited because it is specified outside the program unit containing the dummy argument. If the associated actual argument is an array name, the length inherited by the dummy argument is the length of an array element in the associated actual argument array.

Variables As Dummy Arguments

A dummy argument that is a variable name must be associated with an actual argument that is a variable, array element, substring name, or expression.

You can define a dummy argument that is a variable name within a subprogram if the associated actual argument is a variable name, array element name, or character substring name. You must not redefine a dummy argument that is a variable name within a subprogram if the associated actual argument is a constant, name of a constant, function reference, expression involving operators, or expression enclosed in parentheses.

Arrays As Dummy Arguments

A dummy argument that is an array name must be associated with an actual argument that is an array name, an array element name, or a character substring name.

The size of the actual array must be larger than or equal to the declared size of the dummy array. The number and size of the dimensions can differ only for arrays of type character.

If an actual argument is a noncharacter array name, the size of the dummy argument array must not exceed the size of the actual argument array. Each actual argument array element becomes associated with the dummy argument array element of the same subscript value.

If an actual argument is a noncharacter array element name with a subscript value of `asv`, the dummy argument array element with a subscript value of `dsv` becomes associated with the actual argument array element that has a subscript value of `asv + dsv - 1`.

If an actual argument is a character array name, character array element name, or character substring name and begins at a character storage unit `acu` of an array, character storage unit `dcu` of an associated dummy argument array becomes associated with character storage unit `acu + dcu - 1` of the actual array argument.

Procedures As Dummy Arguments

A dummy argument that is identified as a procedure is called a dummy procedure. A dummy procedure can only be associated with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure.

The following example illustrates the use of a dummy procedure:

```
SUBROUTINE ROOTS
EXTERNAL NEG
.
.
X = QUAD(A,B,C,NEG)
.
.
RETURN
END

FUNCTION QUAD(A,B,C,FUNCT)
INTEGER FUNCT
.
.
VAL = FUNCT(A,B,C)
.
.
RETURN
END

FUNCTION NEG(A,B,C)
.
.
RETURN
END
```

Asterisks As Dummy Arguments

A dummy argument that is an asterisk can appear only in the dummy argument list of a **SUBROUTINE** statement or an **ENTRY** statement in a subroutine subprogram. The corresponding actual argument must be an alternate return specifier.

Recursion

XL FORTRAN allows recursion within your program if you specify the **RECUR** compiler option. This means that procedures can call themselves, either directly or through a chain of other procedures. A main program cannot call itself recursively.

Local variables are static by default, but you can make them automatic by using an **AUTOMATIC** statement or an **IMPLICIT AUTOMATIC** statement.

BLOCK DATA Statement — Block Data Subprogram

name
is the name of the block data subprogram.

A block data subprogram is a program unit that provides initial values for variables and array elements in named common blocks.

The first statement of a block data subprogram must be a **BLOCK DATA** statement. The only other statements that can appear in a block data subprogram are **DIMENSION**, **EQUIVALENCE**, **COMMON**, explicit type, **IMPLICIT**, **PARAMETER**, **SAVE**, **DATA**, **END**, and comment lines.

You can have more than one block data subprogram in an executable program, but only one can be unnamed.

More than one named common block can be initialized in a block data subprogram.

Restrictions on common blocks in block data subprograms are:

- All items in a named common block must appear in the **COMMON** statement even though they are not all initialized.
- The same named common block must not be referenced in two different block data subprograms.
- Only items in named common blocks can be initialized in block data subprograms.

The name, if given, must not be the same as another subprogram, entry, main program, or common block in the executable program. It also must not be the same as a data item in this subprogram. **DATA** statements and explicit type statements initializing items can follow or precede the definitions of those items in named **COMMON** statements.

Local variables cannot be declared in a **BLOCK DATA** subprogram. A variable or array made equivalent to another variable or array in a common block is considered to be in that common block.

Chapter 11. Input/Output Statements

This chapter describes:

- Records
- Files
- Units
- The input/output statements: **READ**, **WRITE**, **PRINT**, **OPEN**, **CLOSE**, **INQUIRE**, **BACKSPACE**, **ENDFILE**, and **REWIND**
- **IOSTAT** values.

Records

A record is a sequence of characters or a sequence of values. The three kinds of records are formatted, unformatted, and endfile.

Formatted Records

A formatted record is a sequence of any ASCII characters. When a formatted record is read, data values represented by characters are converted to an internal form. When a formatted record is written, the data to be written is converted from internal form to characters.

If a formatted record is printed using the AIX **asa** command[1] (see Appendix E of the *User's Guide for IBM AIX XL FORTRAN Compiler/6000*, SC09–1257–00 for information on this command), the first character of the record determines vertical spacing and is not printed. The remaining characters of the record, if any, are printed beginning at the left margin. Vertical spacing can be specified in a format specification in the form of literal data. Vertical spacing is as follows:

First Character of Record	Vertical Spacing Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

The characters and spacing shown are those defined for print records. If you use any other character as the first character of the record, it will be ignored. If the print record contains no characters, spacing is advanced by one line and a blank line is printed. If records are to be displayed at a terminal, control characters are also employed, but only the characters blank and zero produce the spacing shown. (The AIX **asa** command must be used if these print codes are to be displayed on a terminal.)

Unformatted Records

An unformatted record is a sequence of values in a system dependent form and can contain both character and noncharacter data or can contain no data. The values are in their internal form and are not converted in any way when read or written.

---

1. Printing can be performed on a printer or on some other device.

Endfile Records

If it exists, an endfile record is the last record of a file. It can be written by an **ENDFILE** statement and has no length.

Files

A file is a sequence of records. The two kinds of files are external and internal. Access to an external file can be sequential or direct.

External Files

An external file is a file stored on an input/output device such as a disk, tape, or terminal.

An external file is said to exist for a program if it is available to the program for reading or was created within the program. Creating an external file causes it to exist when it did not previously. Deleting an external file ends its existence. An external file can exist but contain no records. All input/output statements can refer to external files that exist, or to external files that do not exist.

An external file can have a name. The name is system dependent.

Note: A valid AIX file name must have a full path name of total length ≤2048 characters, with each file name ≤256 characters long (though the full path name need not be specified).

The position of an external file is usually established by the preceding input/output operation. An external file can be positioned to:

- An initial point, which is the position just before the first record.
- A terminal point, which is the position just after the last record.
- A current record, when the file is positioned within a record. Otherwise, there is no current record.
- A preceding record, which is the record just before the current file position. A preceding record does not exist when the file is positioned at its initial point or at the first record of the file.
- A next record, which is the record just after the current file position. The next record does not exist when the file is positioned at the terminal point or in the last record of the file.
- An indeterminate position after an error.

External File Access — Sequential or Direct

The two methods of accessing the records of an external file are sequential and direct. The method is determined when the file is connected to a unit.

A file connected for sequential access contains records in the order they were written. The records must be either all formatted or all unformatted; the last record of the file must be an endfile record. The records must not be read or written by direct access input/output statements during the time the file is connected for sequential access.

The records of a file connected for direct access can be read or written in any order. The records must be either all formatted or all unformatted; the last record of the file can be an endfile record if the file was previously connected for sequential access. In this case, the endfile record is not considered a part of the file when it is connected for direct access. The records must not be read or written by sequential access input/output statements or read or written using list–directed formatting during the time the file is connected for direct access.

Each record in a file connected for direct access has a record number that identifies its order in the file. The record number is an integer value that must be specified when the record is read or written. Records are numbered sequentially. The first record is number 1. Records need not be read or written in the order of their record numbers. For example, records 9, 5, and 11 can be written in that order without writing the intermediate records.

All records in a file connected for direct access must have the same length, which is specified in the **OPEN** statement (see "OPEN Statement" on page 106) when the file is connected.

Records in a file connected for direct access cannot be deleted but can be rewritten with a new value. A record cannot be read, after the end of the file, unless it is first written.

Internal Files

An internal file is a character variable, character array, character array element, or character substring.

If an internal file is a character variable, character array element, or character substring, the file consists of one record with a length equal to that of the variable, array element, or substring. If an internal file is a character array, each element of the array is a record of the file, with each record having the same length.

Reading and writing records are accomplished by sequential–access formatted input/output statements. **READ** and **WRITE** are the only input/output statements that can specify an internal file.

If a **WRITE** statement writes less than an entire record, blanks fill the remainder of the record.

An internal file always exists.

A variable, array element, or character substring that is a record of an internal file can become defined or undefined by means other than an output statement. For example, you can define it by a character assignment statement.

Units

A unit is a means of referring to an external file. Programs refer to external files by the unit numbers specified in unit specifiers in input/output statements. See page 101 for the form of a unit specifier.

Connection of a Unit

The association of a unit with an external file is called a connection. Connection must occur before the records of the file can be read or written. Connection can occur by preconnection, which is prior to running the program, or by an **OPEN** statement.

Units 0, 5, and 6 are preconnected when the program starts:

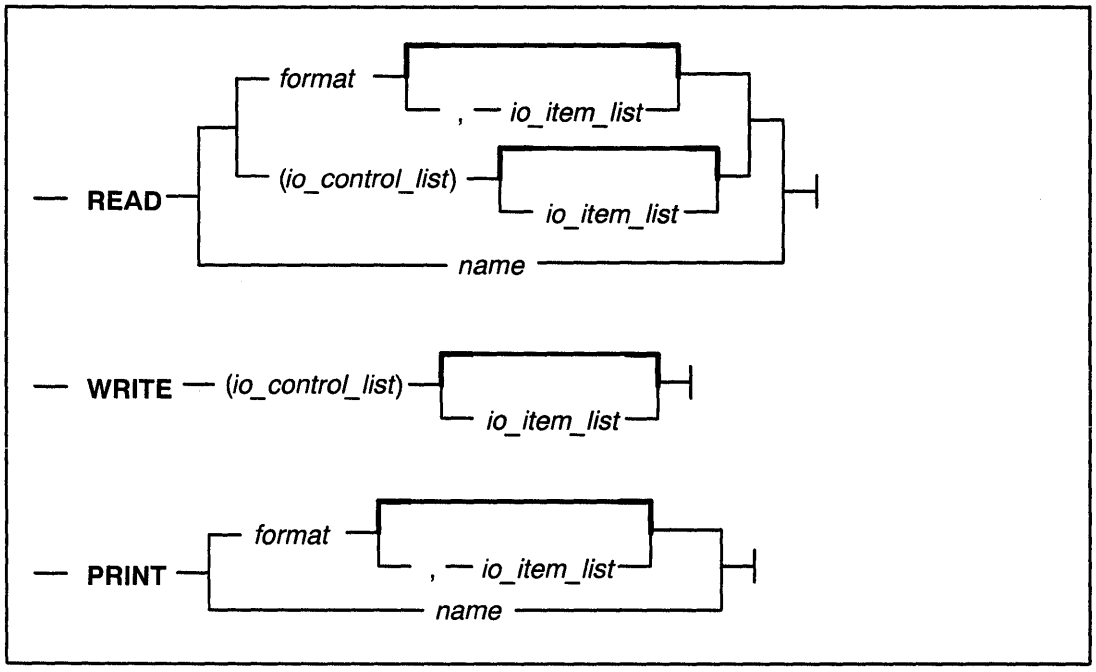
- Unit 0 is connected to the standard error.
- Unit 5 is connected to the standard input for sequential formatted input/output.
- Unit 6 is connected to the standard output for sequential formatted input/output.

All other units are also preconnected (unit n to a file named **fort.***n*). These files need not exist and are created if you use their units without first performing an **OPEN** statement. The default connection is for sequential formatted input/output.

A file can be connected and not exist, for example, a preconnected new file.

The **CLOSE** statement disconnects a file from a unit. The file can be connected again within the same program to the same unit or to a different unit, and the unit can be connected again within the same program to the same file or to a different file.

READ, WRITE, and PRINT Statements



format
is a format identifier, described below under **FMT**=*format*.

name
is a **NAMELIST** name.

io_item
is an input/output list item. An input/output list specifies the data to be transferred. An input/output list item can be:

- A variable name.
- An array element name.
- A character substring name.
- An array name (cannot be an assumed–size array). The array is treated as if all of its elements were specified in the order they are arranged in storage.
- (In an output list only.) Any other expression except a character expression involving concatenation of an operand whose length specifier is an asterisk in parentheses (indicating inherited length) unless the operand is the name of a constant. A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.
- An implied–DO list, described on page 105.

io_control
is a list that must contain one unit specifier, and can also contain one of each of the other valid specifiers. The valid specifiers are:

[**UNIT=**] *u*
is a unit specifier that specifies the unit to be used in the input/output operation. *u* is an external unit identifier or internal file identifier.

An external unit identifier refers to an external file. It is one of the following:

- An **INTEGER*4** expression whose value is 0 through 99.
- An asterisk, representing standard input or standard output.

  Note: Although other input/output statements also allow a unit specifier, only the **READ** and **WRITE** statements allow its value to be an asterisk.

An internal file identifier refers to an internal file. It is the name of a character variable, character array, character array element, or character substring.

If the optional characters **UNIT=** are omitted, *u* must be the first item in *io_control_list*.

[**FMT=**] *format*
is a format specifier that specifies the format to be used in the input/output operation. *format* is a format identifier, that can be:

- The statement label of a **FORMAT** statement. (The **FORMAT** statement is described on page 117.) The **FORMAT** statement must be in the same program unit.
- The name of an **INTEGER*4** variable that was assigned the statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same program unit.
- A character constant delimited by apostrophes or double quotation marks. It must begin with a left parenthesis and end with a right parenthesis. Only the format codes described in the **FORMAT** statement can be used between the parentheses. Blank characters can precede the left parenthesis, or follow the right parenthesis. An apostrophe or double quotation mark in a constant is represented by two consecutive apostrophes or double quotation marks respectively.
- A character variable or character array element that contains character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes described in the **FORMAT** statement can be used between the parentheses. Blank characters can precede the left parenthesis, or follow the right parenthesis. The length of the format identifier must not exceed the length of the array element.
- The name of a character array. (See "Character Format Specification" on page 120 for more information.) The character array name must contain character data whose leftmost characters constitute a valid format identifier. The length of the format identifier can exceed the length of the first array element. The format specifier is considered the concatenation of all the array elements of the array in the order given by array element ordering.
- An array name of type integer, real, double precision, logical, complex, or double complex. The data must be a valid format identifier as described under character array.
- Any character expression except one involving concatenation of an operand whose length specifier is an asterisk in parentheses (indicating inherited length) unless the operand is the name of a constant. (See "Character Format Specification" on page 120 for more information.)

- An asterisk, specifying list–directed formatting. (List–directed formatting is described on page 134 .)

If the optional characters **FMT=** are omitted, *format* must be the second item in *io_control_list*, and the first item must be the unit specifier.

REC= *integer_expr*
is a record specifier that specifies the number of the record to be read or written in a file connected for direct access. The **REC=** specifier is only permitted for direct input/output. *integer_expr* is an integer expression whose value is positive. A record specifier is not valid if formatting is list–directed, if the unit specifier specifies an internal file, or if an end–of–file specifier is specified. The record specifier represents the relative position of a record within a file. The relative position number of the first record is 1.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. ios is the name of a variable or array element of type **INTEGER*4**. When the input/output statement containing this specifier finishes execution, ios is defined with:

- A negative value if an end–of–file condition was encountered and no error occurred while the input/output statement was executing, or if the **READ** or **WRITE** statement follows the **ENDFILE** statement.
- A zero value if the input/output operation completes without any errors.
- A positive value if an error occurred during the input/output operation. The meaning of a positive value is system dependent. (See "IOSTAT Values" on page 114 for further information.)

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same program unit to which control is to transfer in the case of an error. If you do not specify **ERR=** or **IOSTAT=**, the program stops when an error is detected.

END= *stmt_label*
is an end–of–file specifier that specifies a statement label at which the program is to continue when an endfile record is encountered while a file is being read, and no error occurred. Coding the **ERR** parameter suppresses the error message for end–of–file. This specifier can only be specified in a **READ** statement that refers to a unit connected for sequential access. If you stipulate an end–of–file specifier, you cannot have a record specifier.

NUM= *integer_variable*
is a number specifier that specifies the number of bytes of data transmitted from the elements specified by the input/output list. *integer_variable* is the name of a variable or array element of type **INTEGER*4**. The **NUM=** specifier is only permitted for unformatted input/output. Coding the **NUM** parameter suppresses the indication of an error that would occur if the number of bytes represented by the input/output list is greater than the number of bytes that can be written into the record. In this case, *integer_variable* is set to a value that is the maximum length record that can be written. Data from remaining input/output list items is not written into subsequent records.

[**NML=**] *name*
is a **NAMELIST** specifier that specifies the name of a **NAMELIST** that you have previously defined. If the optional characters **NML=** are not specified, the **NAMELIST** name must appear as the second parameter in the list. If both **NML=** and **UNIT=** are specified, all the parameters can appear in any order. The **NML=** specifier is an alternative to **FMT=**. Both **NML=** and **FMT=** cannot be specified in the same input/output statement.

A **READ** statement without *io_control_list* specified specifies the same unit as a **READ** statement with *io_control_list* specified in which the external unit identifier is an asterisk.

Transfer is made to the statement specified by the **ERR=** parameter if an error is detected. If **IOSTAT=** is specified, a positive integer value is assigned to *ios* when an error is detected. The program then continues with the statement specified with the **ERR=** parameter, if present, or with the next statement if **ERR=** is not specified. If the **ERR=** and **IOSTAT=** parameters are both omitted, the program stops if an error is detected. If you do not use the **END=** specifier, and the end–of–file is encountered, execution continues at the next statement.

`PRINT format` has the same effect as a `WRITE(*,format)`.

Categories of READ, WRITE, and PRINT Statements

A **READ** or **WRITE** statement can be a formatted input/output statement or an unformatted input/output statement. The **PRINT** statement is a formatted input/output statement.

A formatted input/output statement contains a format identifier and transfers data with editing (conversion) occurring between the internal form of the data and the character representation of that data in records. The three methods of formatting are:

- Format–directed formatting, where editing is controlled by edit descriptors in a format specification. Format specifications are described on page 117.
- List–directed formatting, where editing is controlled by the types and lengths of the data being read or written. List–directed formatting is described on page 134.

  If a formatted **READ**, **WRITE**, or **PRINT** statement has an asterisk as a format identifier, the statement is a list–directed input/output statement, and a record specifier must not be present.
- **NAMELIST** formatting, where editing is controlled by a **NAMELIST** list. **NAMELIST** formatting is described on page 136.

An unformatted input/output statement does not contain a format identifier and transfers data without performing editing.

A **READ** or **WRITE** statement is a direct access input/output statement if it contains a record specifier, or a sequential access input/output statement if it does not contain a record specifier.

Executing READ, WRITE, and PRINT Statements

The **READ** statement obtains data from an external or internal file and places it in internal storage. Values are transferred from the file to the data items specified by the input list (*io_item_list*), if one is specified.

The **WRITE** statement places data obtained from internal storage into an external or internal file. The **PRINT** statement places data obtained from internal storage into an external file.Values are transferred to the file from the data items specified by the output list (*io_item_list*) and format specification, if they are specified. Execution of a **WRITE** or **PRINT** statement for a file that does not exist creates the file, unless an error occurs.

If the *io_item_list* is omitted (in a **PRINT** statement), a blank record is transmitted to the output device unless the **FORMAT** statement referred to contains as its first specification a character constant or slashes. In this case, the records indicated by these specifications are transmitted to the output device.

If a transmission error is detected during a **READ** or **WRITE** statement, control is transferred to the statement specified by **ERR=**. No indication is given of which record or records could not be read or written; only that the error occurred during transmission of data. If **IOSTAT** is specified, a positive integer value is assigned to *ios* when the error is detected. If **ERR=** is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause **IOSTAT** to be set and transfer will not be made to the statement specified by **ERR=**.

File Position Before and After Data Transfer

The FORTRAN standards do not specify the initial position of a file that is explicitly opened for sequential input/output, so the following conventions have been adopted for XL FORTRAN:

On an explicit **OPEN** (by an **OPEN** statement):

- If the file **STATUS** is **NEW** or **SCRATCH**, the file is positioned at the beginning.
- If **STATUS='OLD'** is specified, the file is positioned at the end.
  - If the next operation is a **WRITE**, it will append to the file.
  - If the next operation is a **READ**, the file is repositioned to the beginning so that the first record is read.

The implementation of an implicit **OPEN** is equivalent to an explicit **OPEN** with **STATUS='NEW'** (that is, the file is positioned at the beginning). Thus:

- If the first input/output operation on the file is a **READ**, it will read the first record of the file.
- If the first input/output operation on the file is a **WRITE**, it will delete the contents of the file and write at the first record.

Therefore, to append to an existing file, the file must be explicitly opened with an **OPEN** statement with **STATUS='OLD'** specified before a **WRITE** statement can execute.

If you do not perform an explicit **OPEN**, the implicit **OPEN** will open a default file.

A **REWIND** statement can be used to position a file at its beginning. The preconnected units 5 and 6 are positioned as they come from the program's parent process.

The positioning of a file prior to data transfer depends on the method of access:

- Sequential access for an external file: On input, the file is positioned at the beginning of the next record. This record becomes the current record. On output, a new record is created and becomes the last record of the file.

  Sequential access for an internal file: The file is positioned at the beginning of the first record of the file. This record becomes the current record.

- Direct access: The file is positioned at the beginning of the record specified by the record specifier. This record becomes the current record.

After data transfer, the file is positioned:

- Beyond the endfile record if an end–of–file condition exists as a result of reading an endfile record.

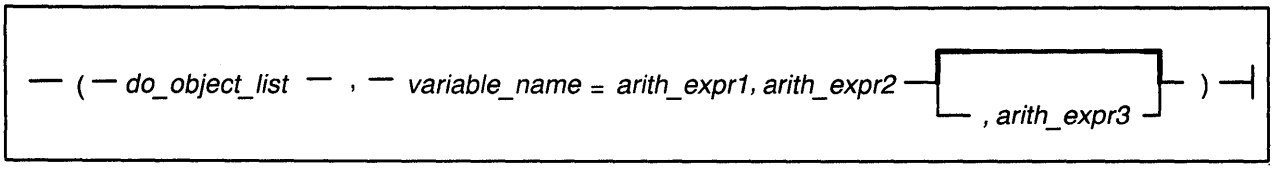
- Beyond the last record read or written if no error or end–of–file condition exists. That last record becomes the preceding record. A record written on a file connected for sequential access becomes the last record of the file.

If a file is positioned beyond the endfile record, a **READ**, **WRITE**, **PRINT**, or **ENDFILE** statement cannot execute. A **BACKSPACE** or **REWIND** statement can be used to reposition the file.

If an error occurs, the position of an external file is indeterminate.

Implied–DO List in a READ, WRITE, or PRINT Statement

An implied–**DO** list can be used in a **READ**, **WRITE**, or **PRINT** statement to specify the data to be transferred. Its form is:



do_object
is an input/output list item. (See "*io_item*" on page 100.)

variable_name
arith_expr1
arith_expr2
arith_expr3
are as specified for the **DO** statement. (See page 74.)

The range of an implied–**DO** list is the list *do_object_list*. The iteration count and the values of the **DO** variable are established from *arith_expr1*, *arith_expr2*, and *arith_expr3*, the same as for a **DO** statement. (See "Executing a DO Statement" on page 75.) When the implied–**DO** list is executed, the items in the *do_object_list* are specified once for each iteration of the implied–**DO** list, with the appropriate substitution of values for any occurrence of the **DO** variable.

In a **READ** statement, the **DO** variable or an associated data item must not appear as an input list item in the *do_object_list*, but can be read in the same **READ** statement outside of the implied–**DO** list. For example:

```
      READ(3,150) ISIZE,(LIST(I),I=1,ISIZE)
150   FORMAT(10I7)
```

In the example, the value of `ISIZE` is read with the same **READ** statement but outside of the implied–**DO** list of which it is a part. One element of the array `LIST` is defined with each iteration of the implied–**DO** list.

Examples of READ, WRITE, and PRINT Statements

Example of Formatted READ and WRITE Statements

```
      INTEGER LENGTH,WIDTH,DEPTH
      CHARACTER*8 CHR_TIME
      .
      .
      READ(10,200) LENGTH,WIDTH,DEPTH
200   FORMAT(I5,I10,I10)
      WRITE(*,'(A,A)') 'The time is:',CHR_TIME(1:8)
```

Example of Unformatted READ and WRITE Statements

```
INTEGER DATA_UNIT,SIZE,A(1000),BUFFER(2000)
.
.
READ(UNIT=DATA_UNIT) SIZE,(A(J),J=1,SIZE)
WRITE(20) BUFFER
```

OPEN Statement

open_list
is a list that must contain one unit specifier (**UNIT=** *u*) and can also contain one of each of the other valid specifiers. The valid specifiers are:

[UNIT=] *u*
is a unit specifier in which *u* must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by an **INTEGER*4** expression, whose value is 0 through 99. If the optional characters **UNIT=** are omitted, *u* must be the first item in *open_list*.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. *ios* is the name of a variable or array element of type **INTEGER*4**. When the input/output statement containing this specifier finishes execution, *ios* is defined with a zero value if the input/output operation completed without any errors, and a positive value if an error occurred during the input/output operation. If you do not specify **ERR=** or **IOSTAT=**, the program stops when an error is detected. (See "IOSTAT Values" on page 114 for further information.)

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same program unit to which control is to transfer in the case of an error. If you do not specify **ERR=** or **IOSTAT=**, the program stops when an error is detected.

FILE= *char_expr*
is a file specifier, that specifies the name of the file to be connected to the specified unit. *char_expr* is a character expression whose value, when any trailing blanks are removed, is a valid AIX file name. If the file specifier is omitted, the unit becomes connected to **fort.***u* where *u* is the unit specified with any leading blanks removed.

Note: A valid AIX file name must have a full path name of total length ≤2048 characters, with each file name ≤256 characters long (though the full path name need not be specified).

STATUS= *char_expr*
specifies the status of the file when it is opened. *char_expr* is a character expression whose value, when any trailing blanks are removed, is one of the following:

- **OLD**, to connect an existing file to a unit. If **OLD** is specified, a file specifier must be specified, and the file must exist.
- **NEW**, to create a new file and connect it to a unit. If **NEW** is specified, a file specifier must be specified, and the file must not exist.
- **SCRATCH**, to create and connect a new file that will be deleted when it is disconnected. **SCRATCH** must not be specified with a named file (that is, **FILE=** *char_expr* must be omitted).
- **UNKNOWN**, to connect an existing file, or to create and connect a new file. If the file exists, it is connected as **OLD**. If the file does not exist, it is connected as **NEW**.

UNKNOWN is the default.

ACCESS= *char_expr*
specifies the access method for the connection of the file. *char_expr* is a character expression whose value, when any trailing blanks are removed, is either **SEQUENTIAL** or **DIRECT**. **SEQUENTIAL** is the default. If **ACCESS** is **DIRECT**, **RECL=** must be specified. If **ACCESS** is **SEQUENTIAL**, **RECL=** must not be specified.

FORM= *char_expr*
specifies whether the file is connected for formatted or unformatted input/output. *char_expr* is a character expression whose value, when any trailing blanks are removed, is either **FORMATTED** or **UNFORMATTED**. If the file is being connected for sequential access, **FORMATTED** is the default. If the file is being connected for direct access, **UNFORMATTED** is the default.

RECL= *integer_expr*
specifies the length of each record in a file being connected for direct access. *integer_expr* is an **INTEGER*4** expression whose value must be positive. This specifier must be omitted when a file is being connected for sequential access.

BLANK= *char_expr*
controls the default interpretation of blanks when using a format specification. *char_expr* is a character expression whose value, when any trailing blanks are removed, is either **NULL** or **ZERO**. See "BN (Blank Null) and BZ (Blank Zero) Editing" on page 124 for descriptions of **NULL** and **ZERO**. If **BLANK** is specified, you must use **FORM= 'FORMATTED'**. If **BLANK** is not specified and you specify **FORM='FORMATTED'**, **NULL** is the default.

The **OPEN** statement can be used to connect an existing external file to a unit, create an external file that is preconnected, create an external file and connect it to a unit, or change certain specifiers of a connection between an external file and a unit.

If a unit is connected to a file that exists, an **OPEN** statement for that unit cannot be performed. If the file specifier is not included in the **OPEN** statement, the file to be connected to the unit is the same as the file to which the unit is connected.

If the file to be connected to the unit does not exist, but is the same as the file to which the unit is preconnected, the properties specified by the **OPEN** statement become a part of the connection.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a **CLOSE** statement without a **STATUS=** *char_expr* specifier had been executed for the unit immediately prior to the execution of the **OPEN** statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the **BLANK=** *char_expr* specifier can have a value different from the one currently in

effect. Execution of the **OPEN** statement causes the new value of the **BLANK=** *char_expr* specifier to be in effect. The position of the file is unaffected.

If a file is connected to a unit, an **OPEN** statement on that file and a different unit cannot be performed.

Example of the OPEN Statement

```
CHARACTER*20 FNAME
FNAME = 'INPUT.DAT'
OPEN(UNIT=8,FILE=FNAME,STATUS='NEW',FORM='FORMATTED')
```

CLOSE Statement

close_list
is a list that must contain one unit specifier (**UNIT=***u*) and can also contain one of each of the other valid specifiers. The valid specifiers are:

[UNIT=] *u*
is a unit specifier in which u must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by an **INTEGER*4** expression, whose value is 0 through 99. If the optional characters **UNIT=** are omitted, *u* must be the first item in *close_list*.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. *ios* is the name of a variable or array element of type **INTEGER*4**. When the input/output statement containing this specifier finishes executing, *ios* is defined with a zero value if the input/output operation completed without any errors, and a positive value if an error occurred during the input/output operation. If you do not specify **ERR=** or **IOSTAT=**, the program stops when an error is detected. (See "IOSTAT Values" on page 114 for further information.)

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same program unit to which control is to transfer in the case of an error. If you do not specify **ERR=** or **IOSTAT=**, the program stops when an error is detected.

STATUS= *char_expr*
specifies the status of the file after it is closed. *char_expr* is a character expression whose value, when any trailing blanks are removed, is either **KEEP** or **DELETE**.

- If **KEEP** is specified for a file that exists, the file will continue to exist after the **CLOSE** statement. If **KEEP** is specified for a file that does not exist, the file will not exist after the **CLOSE** statement. **KEEP** must not be specified for a file whose status prior to executing the **CLOSE** statement is **SCRATCH**.
- If **DELETE** is specified, the file will not exist after the **CLOSE** statement.

The **CLOSE** statement disconnects an external file from a unit.

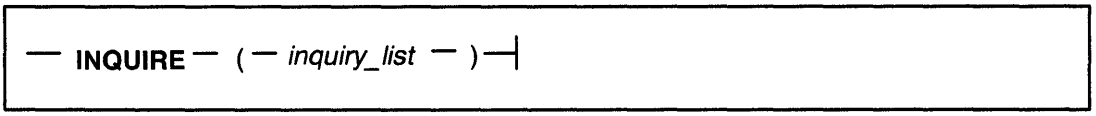
A **CLOSE** statement that refers to a unit can occur in any program unit of an executable program and need not occur in the same program unit as the **OPEN** statement referring to that unit. XL FORTRAN will execute a **CLOSE** statement specifying a unit that does not exist or has no file connected; the **CLOSE** statement has no effect in this case.

When an executable program stops for reasons other than an error condition, all units that are connected are closed. Each unit is closed with status **KEEP** unless the file status prior to completion was **SCRATCH**, in which case the unit is closed with status **DELETE**. The effect is as though a **CLOSE** statement without a **STATUS=** *char_expr* specifier were executed on each connected unit.

Examples of the CLOSE Statement

```
CLOSE(15)
CLOSE(UNIT=16,STATUS='DELETE')
```

INQUIRE Statement



inquiry_list
is a list of inquiry specifiers. In an **INQUIRE**–by–file statement, *inquiry_list* must contain one file specifier (**FILE=** *char_expr*), must not contain a unit specifier (**UNIT=** *u*), and can contain at most one of each of the other inquiry specifiers. In an **INQUIRE**–by–unit statement, *inquiry_list* must contain one unit specifier, must not contain a file specifier, and can contain at most one of each of the other inquiry specifiers. The inquiry specifiers are:

FILE= *char_expr*
is a file specifier. It specifies the name of the file about which an **INQUIRE**–by–file statement is inquiring. *char_expr* is a character expression whose value, when any trailing blanks are removed, is a valid AIX file name. The named file does not have to exist, nor does it have to be associated with a unit.

Note: A valid AIX file name must have a full path name of total length ≤2048 characters, with each file name ≤256 characters long (though the full path name need not be specified).

[UNIT=] *u*
is a unit specifier. It specifies the unit about which an **INQUIRE**–by–unit statement is inquiring. *u* must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by an **INTEGER*4** expression, whose value is 0 through 99. If the optional characters **UNIT=** are omitted, *u* must be the first item in *inquiry_list*.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. *ios* is the name of a variable or array element of type **INTEGER*4**. When the input/output statement containing this specifier finishes executing, *ios* is defined with a zero value if the input/output operation completed without any errors, and a positive value if an error occurred during the input/output operation. (See "IOSTAT Values" on page 114 for further information.)

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same program unit to which control is to transfer in the case of an error.

EXIST= *ex*
indicates if a file or unit exists. *ex* is a logical variable or logical array element of length 4 that is assigned the value `true` or `false`. For an **INQUIRE**–by–file statement, the value `true` is assigned if the file specified by **FILE=** *char_expr* exists. The value `false` is assigned if the file does not exist. For an **INQUIRE**–by–unit statement, the value `true` is assigned if the unit specified by **UNIT=** *u* exists. The value `false` is assigned if the unit does not exist.

OPENED= *od*
indicates if a file or unit is connected. *od* is a logical variable or logical array element of length 4 that is assigned the value `true` or `false`. For an **INQUIRE**–by–file statement, the value `true` is assigned if the file specified by **FILE=** *char_expr* is connected to a unit. The value `false` is assigned if the file is not connected to a unit. For an **INQUIRE**–by–unit statement, the value `true` is assigned if the unit specified by **UNIT=** *u* is connected to a file. The value `false` is assigned if the unit is not connected to a file.

NUMBER= *num*
indicates the external unit identifier currently associated with the file. *num* is an **INTEGER*4** variable or array element that is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, *num* becomes undefined.

NAMED= *nmd*
indicates if the file has a name. *nmd* is a logical variable or logical array element of length 4 that is assigned the value `true` if the file has a name, or the value `false` if the file does not have a name.

NAME= *fn*
indicates the name of the file. *fn* is a character variable or character array element that is assigned the value of the name of the file if the file has a name, or becomes undefined if the file does not have a name.

ACCESS= *char_expr*
indicates whether the file is connected for sequential access or direct access. *char_expr* is a character variable or character array element that is assigned the value **SEQUENTIAL** if the file is connected for sequential access, or the value **DIRECT** if the file is connected for direct access. If there is no connection, *char_expr* becomes undefined.

SEQUENTIAL= *seq*
indicates if the file is connected for sequential access. *seq* is a character variable or character array element that is assigned the value **YES** if the file can be accessed sequentially, the value **NO** if the file cannot be accessed sequentially, or the value **UNKNOWN** if it cannot be determined.

DIRECT= *dir*
indicates if the file is connected for direct access. *dir* is a character variable or character array element that is assigned the value **YES** if the file can be accessed directly, the value **NO** if the file cannot be accessed directly, or the value **UNKNOWN** if it cannot be determined.

FORM= *char_expr*
indicates whether the file is connected for formatted or unformatted input/output. *char_expr* is a character variable or character array element that is assigned the value **FORMATTED** if the file is connected for formatted input/output, or the value **UNFORMATTED** if the file is connected for unformatted input/output. If there is no connection, *char_expr* becomes undefined.

FORMATTED= *fmt*
indicates if the file can be connected for formatted input/output. *fmt* is a character variable or character array element that is assigned the value **YES** if the file can be connected for formatted input/output, the value **NO** if the file cannot be connected for formatted input/output, or the value **UNKNOWN** if it cannot be determined.

UNFORMATTED= *unf*
indicates if the file can be connected for unformatted input/output. *fmt* is a character variable or character array element that is assigned the value **YES** if the file can be connected for unformatted input/output, the value **NO** if the file cannot be connected for unformatted input/output, or the value **UNKNOWN** if it cannot be determined.

RECL= *rcl*
indicates the record length of a file connected for direct access. *rcl* is an **INTEGER*4** variable or array element that is assigned the value of the record length. If there is no connection or if the connection is not for direct access, *rcl* becomes undefined.

NEXTREC= *nr*
indicates where the next record can be read or written on a file connected for direct access. *nr* is an **INTEGER*4** variable or array element that is assigned the value `n + 1`, where `n` is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records were read or written since the connection, *nr* is assigned the value `1`. If the file is not connected for direct access or if the position of the file cannot be determined because of a previous error, *nr* becomes undefined.

BLANK= *char_expr*
indicates the default treatment of blanks for a file connected for formatted input/output. *char_expr* is a character variable or character array element that is assigned the value **NULL** if all blanks in numeric input fields are ignored (as in **BN** editing), or the value **ZERO** if all nonleading blanks are interpreted as zeros (as in **BZ** editing). If there is no connection, or if the connection is not for formatted input/output, *char_expr* becomes undefined.

The **INQUIRE** statement obtains information about:

- The properties of an external file. When the **INQUIRE** statement is used for this purpose, the file specifier (**FILE=** *char_expr*) must be specified, and the statement is called an **INQUIRE**–by–file statement.
- An external file's association with a particular unit. When the **INQUIRE** statement is used for this purpose, the unit specifier (**UNIT=** *u*) must be specified, and the statement is called an **INQUIRE**–by–unit statement.

An **INQUIRE** statement can be executed before, while, or after a file is associated with a unit. Any values assigned as the result of an **INQUIRE** statement are values that are current

at the time the statement is executed. The *inquiry_list* variables or array elements specified by **EXIST=** *ex* and **OPENED=** *od* always become defined.

Execution of an **INQUIRE**–by–file statement causes definition of the *inquiry_list* variables and array elements as follows:

- Variables or array elements specified by **NAMED=** *nmd*, **NAME=** *fn*, **SEQUENTIAL=** *seq*, **DIRECT=** *dir*, **FORMATTED=** *fmt*, and **UNFORMATTED=** *unf* become defined only if the value of *char_expr* is the name of a file that exists; otherwise, the variables or array elements become undefined.
- Variables or array elements specified by **NUMBER=** *num*, **ACCESS=** *char_expr*, **FORM=** *char_expr*, **RECL=** *rcl*, **NEXTREC=** *nr*, and **BLANK=** *char_expr* become defined only if a variable or array element specified by **OPENED=** *od* becomes defined with the value `true`.

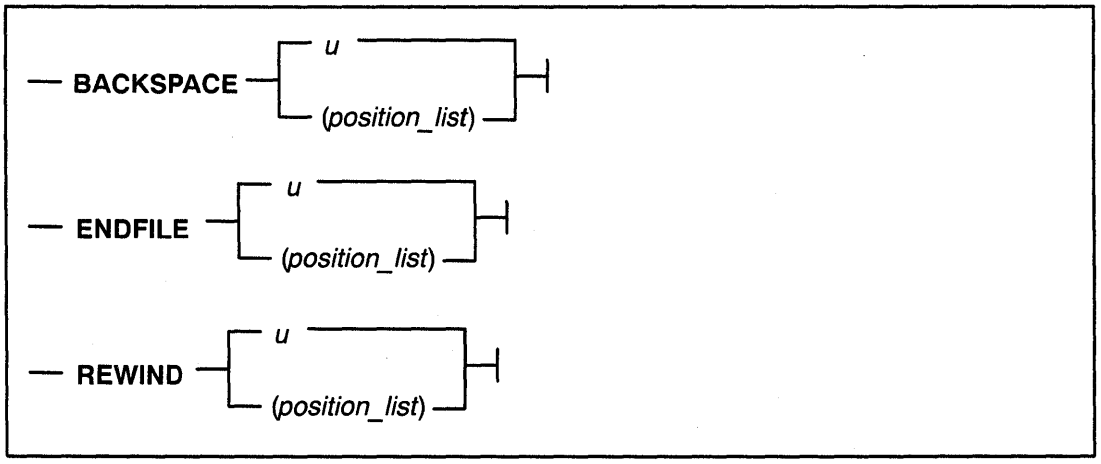
Execution of an **INQUIRE**–by–unit statement causes definition of the *inquiry_list* variables and array elements specified by **NUMBER=** *num*, **NAMED=** *nmd*, **NAME=** *fn*, **ACCESS=** *char_expr*, **SEQUENTIAL=** *seq*, **DIRECT=** *dir*, **FORM=** *char_expr*, **FORMATTED=** *fmt*, **UNFORMATTED=** *unf*, **RECL=** *rcl*, **NEXTREC=** *nr*, and **BLANK=** *char_expr* only if the specified unit exists and if a file is connected to the unit; otherwise, the variables or array elements become undefined.

All value assignments are done according to the rules for assignment statements. Note that no error is given if the value is truncated because the receiving field is too small. The same variable name or array element must not be specified for more than one parameter in the same **INQUIRE** statement, and must not be associated with any other variable or array element in the list of parameters through **EQUIVALENCE**, **COMMON**, or argument passing.

Example of the INQUIRE Statement

```
INQUIRE(FILE=FILE1,EXIST=F_EX,OPENED=F_OD,NUMBER=F_NUM)
```

BACKSPACE, ENDFILE, and REWIND Statements



u
is an external unit identifier. The value of *u* must not be an asterisk.

position_list
is a list that must contain one unit specifier (**UNIT=** *u*) and can also contain one of each of the other valid specifiers. The valid specifiers are:

[**UNIT=**] *u*
: is a unit specifier in which *u* must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by an **INTEGER*4** expression, whose value is 0 through 99. If the optional characters **UNIT=** are omitted, *u* must be the first item in *position_list*.

IOSTAT= *ios*]
: is an input/output status specifier that specifies the status of the input/output operation. *ios* is the name of a variable or array element of type **INTEGER*4**. When the input/output statement containing this specifier finishes executing, *ios* is defined with a zero value if the input/output operation completed without any errors, and a positive value if an error occurred during the input/output operation. (See "IOSTAT Values" on page 114 for further information.)

ERR= *stmt_label*
: is an error specifier that specifies the statement label of an executable statement in the same program unit to which control is to transfer in the case of an error.

External files connected for sequential access can be positioned using **BACKSPACE**, **ENDFILE**, and **REWIND** statements. Transfer is made to the statement specified by the **ERR=** parameter if an error is detected. If **IOSTAT=** is specified, a positive integer value is assigned to *ios* when an error is detected. The program then continues with the statement specified with the **ERR=** parameter, if present, or with the next statement if **ERR=** is not specified. If the **ERR=** and **IOSTAT=** parameters are both omitted, the program stops if an error is detected.

Each command also has the following effects:

- **BACKSPACE** positions a file, connected to a specified unit, before the preceding record. If there is no preceding record, the file position does not change. If the preceding record is the endfile record, the file is positioned before the endfile record. You cannot backspace over records that were written using list–directed formatting, or **NAMELIST** formatting. If you try, the result is unpredictable. You cannot backspace a file that is connected, but does not exist.
- **ENDFILE** writes an endfile record as the next record of a file. This record becomes the last record in the file. If the unit is not connected, an error is detected and transfer is made to the statement specified by the **ERR=** *stmt_label*.
- **REWIND** positions a file at the beginning of the first record of the file. If the unit is not connected, an implicit **OPEN** specifying sequential access is performed to a default file named **fort.***n*, where *n* is the value of *u* with leading zeros removed. If the external file connected to the specified unit does not exist, the **REWIND** statement has no effect. If it does exist, an end–of–file marker is created, if necessary, and the file is positioned at the beginning of the first record. If the file is already positioned at its initial point, the **REWIND** has no effect. The **REWIND** statement causes a subsequent **READ** or **WRITE** statement referring to *u* to read data from or write data into the first record of the external file associated with *u*.

Examples of the BACKSPACE, ENDFILE, and REWIND Statement

```
BACKSPACE 15
BACKSPACE (UNIT=15,ERR=99)
ENDFILE 12
ENDFILE (IOSTAT=IOSS,UNIT=11)
REWIND 9
```

IOSTAT Values

If the error detected is an input/output error, and **IOSTAT** was specified on the input/output statement in error, then the **IOSTAT** variable will be assigned the following value:

- The negative value of the last 3 digits of the message number, if an end–of–file condition exists
- The last 3 digits of the message number, in all other cases.

IOSTAT Value	Description
–1	End of file
–2	End of internal file
6	File not found and STATUS=OLD specified in OPEN
7	Incorrect format of list–directed input in file
8	Incorrect format of list–directed input in internal file
9	Data item too long for the internal file
10	Read error on direct file
11	Write error on direct file
12	Read error on sequential file
13	Write error on sequential file
14	Error opening a file
15	Permanent I/O error encountered on a file
16	Record number invalid for direct I/O
17	I/O statement not allowed for direct file
18	Direct I/O statement on a file not open
19	Unformatted I/O on formatted file
20	Formatted I/O on unformatted file
21	Sequential I/O on direct file
22	Direct I/O on sequential file
23	File connected to another unit
24	OPEN specifiers don't match file attributes
25	RECL not given on OPEN for direct file
26	Negative record length in OPEN
27	OPEN ACCESS specifier invalid
28	OPEN FORMAT specifier invalid
31	OPEN FILE specifier invalid
35	Recursive I/O operation
36	Invalid unit number
38	REWIND error
39	ENDFILE error
40	BACKSPACE error

84	NAMELIST name not found in file
85	NAMELIST name not found in internal file
93	I/O statement not allowed on error unit

All of these errors will cause the **IOSTAT** variable to be set if the input/output status specifier appears in the input/output statement. If **ERR=** or **END=** has been specified, the program will then branch to the indicated statement label. If none of these has been specified, the program will terminate.

Other error messages can be issued by the IBM AIX XL FORTRAN Run Time Environment, but they will not set the **IOSTAT** variable. Depending upon the severity of the error, the program may branch to the location indicated by the **ERR=** specifier. Error messages that do not set the **IOSTAT** value will not cause the program to terminate.

Chapter 12. Input/Output Formatting

Formatted **READ**, **WRITE**, and **PRINT** statements use formatting information to direct the editing (conversion) between internal data representations and character representations in formatted records. (See "Formatted Records" on page 97.) This chapter describes the three methods of formatting:

- Format–directed formatting
- List–directed formatting
- **NAMELIST** formatting.

Format–Directed Formatting

In format–directed formatting, editing is controlled by edit descriptors in a format specification. A format specification is specified in a **FORMAT** statement or as the value of a character array or character expression in a **READ**, **WRITE**, or **PRINT** statement.

FORMAT Statement

format_spec
is described below.

When a format identifier (page 101) in a formatted **READ**, **WRITE**, or **PRINT** statement is a statement label or a statement label assigned to a variable name, the statement label identifies a **FORMAT** statement.

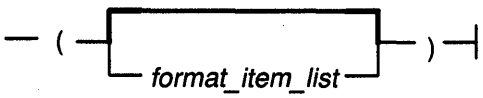
The **FORMAT** statement must have a statement label. You cannot code **FORMAT** statements in **BLOCK DATA** subprograms.

Examples of the FORMAT Statement

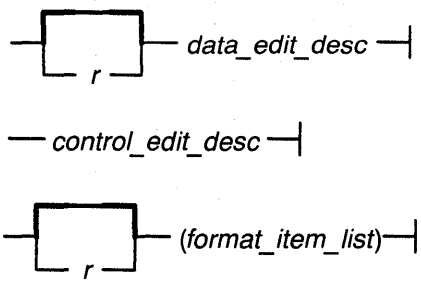
```
990   FORMAT(I5, 2F10.2)
880   FORMAT(I5, F10.2, I5)
```

Format Specification

A format specification (*format_spec*) has the form:



A *format_item* has any of the following forms:



r
is an unsigned, nonzero, integer constant called a repeat specification. The default is 1.

data_edit_desc
is a data (or repeatable) edit descriptor. The forms are:

Forms	Use	See Page
A A*w*	Edits character values	123
E*w.d* E*w.d*E*e* E*w.d*D*e* D*w.d* Q*w.d*	Edits real and complex numbers with exponents	125
F*w.d*	Edits real and complex numbers without exponents	126
G*w.d* G*w.d*E*e* G*w.d*D*e*	Edits integer, real, complex, and logical data fields, with the output format adapting to the magnitude of the number	127
I*w* I*w.m*	Edits integer numbers	129
L*w*	Edits logical values	130
Z*w* Z*w.m*	Edits hexadecimal values	132
O*w.m*	Edits octal values	133
B*w.m*	Edits binary values	133

where:

w is an unsigned, nonzero, integer constant that specifies the width of a field.

m is an unsigned, integer constant that specifies the number of digits to be printed.

d is an unsigned, integer constant that specifies the number of digits to the right of the decimal point.

e is an unsigned, nonzero, integer constant that specifies the number of digits in the exponent field.

control_edit_desc
is a control (or nonrepeatable) edit descriptor. The forms are:

Forms	Use	See Page
/	Specifies the end of data transfer on the current record	122
:	Specifies the end of format control if there are no more items in the input/output list	122
'str' "str"	Specifies a character string (str) for output	123
$	Specifies end–of–record is to be suppressed	123
BN	Specifies nonleading blanks in numeric input fields are to be ignored	124
BZ	Specifies nonleading blanks in numeric input fields are to be interpreted as zeros	124
nHstr	Specifies a character string for output	128
kP	Specifies a scale factor	130
S SS	Specifies plus signs are not to be written	131
SP	Specifies plus signs are to be written	131
Tc	Specifies the absolute position in a record from which, or to which, the next character is transferred	131
TLc	Specifies the relative position (backward from the current position in a record) from which, or to which, the next character is transferred	131
TRc	Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred	131
nX	Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred	131

where:

k is an optionally signed integer constant that specifies the scale factor to be used.

c is an unsigned, nonzero, integer constant that specifies the character position in a record.

n is the number of characters in a literal field.

Commas separate edit descriptors. You can omit the comma between a **P** edit descriptor and an **F**, **E**, **D**, **G**, or **Q** edit descriptor immediately following it, before or after a slash edit descriptor, and before or after a colon edit descriptor. If the comma is omitted in other circumstances, characters in the format specification are skipped until the next valid format code is found.

FORMAT specifications can also be given as character constants or character expressions in input/output statements.

Character Format Specification

When a format identifier (page 101) in a formatted **READ**, **WRITE**, or **PRINT** statement is a character array name or character expression, the value of the array or expression is a character format specification. Such a format specification has the form *format_spec*, described on page 117.

If the format identifier is a character array element name, the format specification must be completely contained within the array element. If the format identifier is a character array name, the format specification can continue beyond the first element into following consecutive elements.

Blanks can precede the format specification. Character data can follow the right parenthesis that ends the format specification, with no effect on the format specification.

Example of Character Format Specification

```
CHARACTER*18 CHARVAR
.
.
CHARVAR = '(F10.2, I5, F10.2)'
WRITE(*,CHARVAR) SOLID, LIQUID, GAS
```

Interaction Between an Input/Output List and a Format Specification

The beginning of format–directed formatting initiates format control. Each action of format control depends on the next edit 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 item, at least one data (repeatable) edit descriptor must exist in the format specification. Note that an empty format specification (parentheses only) can be used only if there are no items in the input/output list. In this case, one input record is skipped or one output record containing no characters is written.

A format specification is interpreted from left to right except when a repeat specification (r) is present. A format item preceded by a repeat specification is processed as a list of r format specifications or edit descriptors identical to the format specification or edit descriptor without the repeat specification.

One item specified by the input/output list corresponds to each data (repeatable) edit descriptor. A list item of type complex requires the interpretation of two **F**, **E**, **D**, **G**, or **Q** edit descriptors. There is no item specified by the input/output list that corresponds to each control (nonrepeatable) edit descriptor. Format control communicates information directly with the record.

Format control operates as follows:

1. If a data (repeatable) edit descriptor is encountered, format control processes an input/output list item, if there is one, or terminates the input/output command if the list is empty. If the list item processed is type complex, two **F**, **E**, **D**, **G**, or **Q** edit descriptors are processed.
2. If a colon edit descriptor is encountered, format control processes an input/output list item, if there is one, or terminates the command if the list is empty.
3. If the end of the format specification is reached, format control terminates if the input/output list is empty, or reverts to the beginning of the format specification terminated by the last preceding right parenthesis. The following items apply when the latter occurs:

- The reused portion of the format specification must contain at least one data (repeatable) edit descriptor.
- If reversion is to a parenthesis that is preceded by a repeat specification, the repeat specification is reused.
- Reversion, of itself, has no effect on the scale factor, on the **S**, **SP**, or **SS** edit descriptors, or on the **BN** or **BZ** edit descriptors.
- If format control reverts, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed.

During a read operation, any unprocessed characters of the record are skipped whenever the next record is read.

It is important to consider the maximum size record allowed on the input/output medium when defining a FORTRAN record by a **FORMAT**. For example, if a FORTRAN record is to be printed, the record should not be longer than the printer's line length. For input, the **FORMAT** should not define a FORTRAN record longer than the actual input record for direct access. For sequential access, the record is assumed to be padded on the right with blanks.

Editing

Editing is performed on fields. A field is the part of a record that is read on input or written on output when format control processes one **I**, **F**, **E**, **D**, **G**, **Q**, **L**, **A**, **Z**, **H**, **O**, **B**, apostrophe, or double quotation mark edit descriptor. The field width is the size of the field in characters.

The **I**, **F**, **E**, **D**, **G**, and **Q** edit descriptors are collectively called numeric edit descriptors and are used to format integer, real, and complex data. The general rules that apply to these edit descriptors are:

- On input:
  - Leading blanks are not significant. The interpretation of other blanks is controlled by the **BLANK=** *char_expr* specifier in the **OPEN** statement and the **BN** and **BZ** edit descriptors. A field of all blanks is considered to be zero. Plus signs are optional.
  - In **F**, **E**, **D**, **G**, and **Q** editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location. The field can have more digits than can be represented internally.

- On output:
  - Characters are right–justified inside the field. Leading blanks are supplied if the editing process produces fewer characters than the field width. If the number of characters is greater than the field width, or if an exponent exceeds its specified length, the entire field is filled with asterisks.
  - A negative value is prefixed with a minus sign. By default, a positive or zero value is unsigned; it can be prefixed with a plus sign, as controlled by the **S**, **SP**, and **SS** edit descriptors.
  - In XL FORTRAN, a `NaN` (not a number) is indicated by "`NaNQ`", "`+NaNQ`", or "`–NaNQ`", and infinity is indicated by "`INF`", "`+INF`", or "`–INF`".

Complex Editing

A complex value is a pair of separate real components. Therefore, complex editing is specified by a pair of **F**, **E**, **D**, **G**, **Q**, **Z**, **O**, or **B** edit descriptors. The first edit descriptor edits the real part of the number, and the second edit descriptor edits the imaginary part of the number. The two edit descriptors can be the same or different. One or more control (nonrepeatable) edit descriptors can be placed between the two edit descriptors, but no data (repeatable) edit descriptors can appear between them.

/ (Slash) Editing

Form:

```
/
```

The slash edit descriptor indicates the end of data transfer on the current record.

When you connect a file for input using sequential access, each slash edit descriptor positions the file at the beginning of the next record.

When you connect a file for output using sequential access, each slash edit descriptor creates a new record and positions the file to write at the start of the new record.

When you connect a file for input or output using direct access, each slash edit descriptor increases the record number by one, and positions the file at the beginning of the record that has that record number.

Examples of Slash Editing on Input

```
500   FORMAT(F6.2 / 2F6.2)
100   FORMAT(I4 / I4 / I4)
```

: (Colon) Editing

Form:

```
:
```

The colon edit descriptor terminates format control (which is discussed on page 120) if no more items are in the input/output list. If more items are in the input/output list when the colon is encountered, the colon is ignored.

Example of Colon Editing

```
10    FORMAT(3(:'Array Value',F10.5)/)
```

$ (Dollar) Editing

Form:

$

The dollar edit descriptor inhibits an end–of–record for a sequential **WRITE** statement. Usually, when the end of a format specification is reached, data transmission of the current record ceases and the file is positioned so that the next input/output operation processes a new record. But, if a dollar sign occurs in the format specification, the automatic end–of–record action is suppressed. Subsequent input/output statements can continue reading from or writing to the same record.

Example of Dollar Editing

A common use for dollar sign editing is to prompt for a response, and read the answer from the same line.

```
      WRITE(*,FMT='($,A)')'Enter your age
      READ(*,FMT='(BN,I3)')IAGE
      WRITE(*,FMT=1000)
1000  FORMAT('Enter your height:  ',$)
      READ(*,FMT='(F6.2)')HEIGHT
```

A (Character) Editing

Forms:

A

A*w*

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field, including blanks. If *w* is not specified, the width of the character field is the length of the corresponding input/output list item.

The **A** edit descriptor directs the editing of character values. The **A** edit descriptor should correspond to an input/output list item of type character.

On input, if *w* is greater than or equal to the length (call it *len*) of the input/output list item, the rightmost *len* characters are taken from the input field. If the specified field width is less than len, the *w* characters are left–justified, with *len–w* trailing blanks added.

On output, if *w* is greater than *len*, the output field consists of *w–len* blanks followed by the len characters from the internal representation. If *w* is less than or equal to *len*, the output field consists of the leftmost *w* characters from the internal representation.

Apostrophe/Double Quotation Mark Editing

Forms:

'character string'

"character string"

The apostrophe/double quotation mark edit descriptor specifies a character string in an output format specification. The width of the output field is the length of the character constant.

Examples of Apostrophe/Double Quotation Mark Editing

```
      ITIME=8
c
      WRITE(*,5) ITIME
5     FORMAT('The value is -- ',I2)          ! The value is -- 8
      WRITE(*,10) ITIME
10    FORMAT(I2,'o''clock')                  ! 8o'clock
      WRITE(*,'(I2,''o''''clock'')') ITIME   ! 8o'clock
c
      WRITE(*,15) ITIME
15    FORMAT("The value is -- ",I2)          ! The value is -- 8
      WRITE(*,20) ITIME
20    FORMAT(I2,"o'clock")                   ! 8o'clock
      WRITE(*,'(I2,"o''clock")') ITIME       ! 8o'clock
```

BN (Blank Null) and BZ (Blank Zero) Editing

Forms:

BN

BZ

The **BN** and **BZ** edit descriptors control the interpretation of nonleading blanks by subsequently processed **I**, **F**, **E**, **D**, **G**, and **Q** edit descriptors. **BN** and **BZ** have effect only on input.

BN specifies that blanks in numeric input fields are to be ignored, and remaining characters are to be interpreted as though right–justified. A field of all blanks has a value of zero.

BZ specifies that nonleading blanks in numeric input fields are to be interpreted as zeros.

The initial setting for blank interpretation is determined by the **OPEN** statement and its **BLANK=** *char_expr* specifier. (See page 106 for syntax.) The initial setting is determined as follows:

- If **OPEN** is not specified, blank interpretation is the same as if **BN** editing was specified.
- If **OPEN** is specified but **BLANK=** *char_expr* is not, blank interpretation is the same as if **BN** editing were specified.
- If **OPEN** is specified and **BLANK=** *char_expr* is specified, blank interpretation is the same as if **BN** editing were specified if the value of *char_expr* is **NULL**, or the same as if **BZ** editing were specified if the value of *char_expr* is **ZERO**.

The initial setting for blank interpretation takes effect at the start of a formatted **READ** statement and stays in effect until a **BN** or **BZ** edit descriptor is encountered or until format control finishes. Whenever a **BN** or **BZ** edit descriptor is encountered, the new setting stays in effect until another **BN** or **BZ** edit descriptor is encountered, or until format control terminates.

E (Real with Exponent), D (Double Precision), and Q (Extended Precision) Editing

Forms:

E*w.d*

E*w.d***E***e*

E*w.d***D***e*

D*w.d*

Q*w.d*

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field.

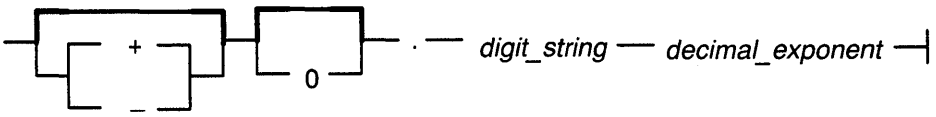
d is an unsigned integer constant that specifies the number of fraction digits to the right of the decimal point.

e is an unsigned, nonzero, integer constant that specifies the number of digits in the output exponent field. *e* has no effect on input.

The **E**, **D**, and **Q** edit descriptors direct editing between real and complex numbers in internal form and their character representations with exponents. An **E**, **D**, or **Q** edit descriptor must correspond to an input/output list item of type real, or to either part (real or imaginary) of an input/output list item of type complex.

The form of the input field is the same as for **F** editing.

The form of the output field for a scale factor of 0 is:



digit_string
is a digit string whose length is the d most significant digits of the value after rounding.

decimal_exponent
is a decimal exponent of one of the following forms (*z* is a digit):

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
E*w.d*	|*decimal_exponent*| ≤ 99	**E**±z_1z_2
E*w.d*	99<|*decimal_exponent*|≤309	±$z_1z_2z_3$
E*w.d***E***e*	|*decimal_exponent*| ≤(10e)–1	**E**±$z_1z_2...z_e$
E*w.d***D***e*	|*decimal_exponent*| ≤(10e)–1	**E**±$z_1z_2...z_e$
D*w.d*	|*decimal_exponent*| ≤ 99	**D**±z_1z_2
D*w.d*	99<|*decimal_exponent*|≤309	±$z_1z_2z_3$
Q*w.d*	|*decimal_exponent*| ≤ 99	**Q**±z_1z_2
Q*w.d*	99<|*decimal_exponent*|≤309	±$z_1z_2z_3$

The scale factor (k; see page 130) controls decimal normalization. If $-d < k \leq 0$, the output field contains $|k|$ leading zeros and $d - |k|$ significant digits after the decimal point. If $0 < k < d + 2$, the output field contains k significant digits to the left of the decimal point and $d - k + 1$ significant digits to the right of the decimal point. You cannot use other values of k.

See page 121 for general information about numeric editing.

Note: If the value to be displayed using the real edit descriptor is outside of the range of representable numbers, XL FORTRAN supports the ANSI/IEEE floating–point format by displaying the following:

Display	Meaning
NaNQ +NaNQ	Positive NaN (not–a–number) Quiet
–NaNQ	Negative NaN (not–a–number) Quiet
NaNS +NaNS	Positive NaN Signaling
–NaNS	Negative NaN Signaling
INF +INF	Positive Infinity
–INF	Negative Infinity

Figure 6. Floating–Point Display

Examples of E, D, and Q Editing on Input

(Assume **BN** editing is in effect for blank interpretation.)

```
Input          Format       Value
12.34          E8.4         12.34
.1234E2        E8.4         12.34
2.E10          E12.6E1      2.E10
```

Examples of E, D, and Q Editing on Output

```
Value          Format       Output
1234.56        E10.3        b0.123E+04
1234.56        D10.3        b0.123D+04
```

F (Real without Exponent) Editing

Form:

F*w.d*

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field.

d is an unsigned integer constant that specifies the number of fraction digits to the right of the decimal point.

The **F** edit descriptor directs editing between real and complex numbers in internal form and their character representations without exponents.

The **F** edit descriptor must correspond to an input/output list item of type real, or to either part (real or imaginary) of an input/output list item of type complex.

The input field for the **F** edit descriptor consists of, in order:

1. An optional sign.
2. A string of digits optionally containing a decimal point. If the decimal point is present, it overrides the *d* specified in the edit descriptor. If the decimal point is omitted, the rightmost *d* digits of the string are interpreted as following the decimal point and leading blanks are converted to zeros if necessary.
3. Optionally, an exponent, having one of the forms:
   - A signed integer constant
   - **E**, **D**, or **Q** followed by zero or more blanks, followed by an optionally signed integer constant. **E**, **D**, and **Q** are processed identically.

The output field for the **F** edit descriptor consists of, in order:

1. Blanks if necessary
2. A minus sign if the internal value is negative, or an optional plus sign if the internal value is zero or positive
3. A string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the scale factor in effect and rounded to *d* fractional digits.

See page 121 for general information about numeric editing.

Examples of F Editing on Input

(Assume **BN** editing is in effect for blank interpretation.)

Input	Format	Value
-100	F6.2	-1.0
2.9	F6.2	2.9
4.E+2	F6.2	400.0

Examples of F Editing on Output

Value	Format	Output
+1.2	F8.4	bb1.2000
.12345	F8.3	bbb0.123

G (General) Editing

Forms:

G*w*.*d*

G*w*.*d***E***e*

G*w*.*d***D***e*

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field.

d is an unsigned integer constant that specifies the number of fraction digits to the right of the decimal point.

e is an unsigned, nonzero, integer constant that specifies the number of digits in the output exponent field.

The **G** edit descriptor is similar to the **E** and **F** edit descriptors except that the output format adapts to the magnitude of the number being edited. Thus the **G** edit descriptor provides a choice of output formats without requiring the magnitude of the numbers to be known ahead of time.

The **G** edit descriptor must correspond to an input/output list item of type real, or to either part (real or imaginary) of an input/output list item of type complex.

G input editing is the same as for **F** editing.

On output, the number is converted using either **E** or **F** editing, depending on the number. The field is padded with blanks on the right as necessary. Letting *N* be the magnitude of the number, editing is as follows:

- If $N < 0.1$ or $N \geq 10^d$:
  - **G***w.d* editing is the same as **E***w.d* editing
  - **G***w.d***E***e* editing is the same as **E***w.d***E***e* editing.
- If $N \geq 0.1$ and $N < 10^d$:
  - **G***w.d* editing is the same as **F***w'.d'* editing, where $w' = w - 4$ and $d' = d - \log_{10}N$
  - **G***w.d***E***e* editing is the same as **F***w'.d'* editing, where $w' = w - (e + 2)$ and $d' = d' - \log_{10}N$.

See page 121 for general information about numeric editing.

Examples of G Editing on Output

Value	Format	Output
1234.56	G12.5	bb1234.6bbbb
123456.	G12.5	b0.12346E+06

H Editing

Form:

*n***H***str*

where:

n is an unsigned, nonzero, integer constant that specifies the number of characters following the **H**, that make up the output field. Blanks are included in the count of characters.

str is a string of any of the characters allowed in a character constant. (See "Character Constants" on page 21.)

The **H** edit descriptor specifies a character string and its length in an output format specification.

If an **H** edit descriptor occurs within a character constant and includes an apostrophe, it must be represented by two consecutive apostrophes, which are counted as one character in specifying *n*.

If an **H** edit descriptor occurs within a character constant and includes double quotation marks, they must be represented by two consecutive double quotation marks, which are counted as one character in specifying *n*.

The **H** edit descriptor must not be used on input.

Examples of H Editing

```
50    FORMAT(16HThe value is —— ,I2)
10    FORMAT(I2,7Ho'clock)
      WRITE(*,'(I2,7Ho''clock)') ITIME
```

I (Integer) Editing

Forms:

I*w*

I*w.m*

where:

w is an unsigned, nonzero, integer constant that specifies the width of the field, including blanks, and the optional sign.

m is an unsigned integer constant that specifies the minimum number of integer digits to be written. *m* must have a value that is less than or equal to *w*. *m* is useful on output only; it has no effect on input.

The **I** edit descriptor directs editing between integers in internal form and character representations of integers. The corresponding input/output list item must be of type integer.

The input field for the **I** edit descriptor must be an optionally signed integer constant, unless it is all blanks. If all blanks, the input field is considered to be zeros.

The output field for the **I** edit descriptor consists of, in order:

1. Zero or more leading blanks
2. A minus sign, if the internal value is negative, or an optional plus sign, if the internal value is zero or positive
3. The magnitude in the form of:
   - If m is not specified, an unsigned integer constant without leading zeros
   - If m is specified, an unsigned integer constant of at least m digits and, if necessary, with leading zeros. If the internal value and m are both zero, blanks are written.

See page 121 for general information about numeric editing.

Examples of I Editing on Input

(Assume **BN** editing is in effect for blank interpretation.)

Input	Format	Value
–123	I6	bb–123
123456	I7.5	b123456
1234	I4	1234

Examples of I Editing on Output

Value	Format	Output
–12	I7.6	–000012
12345	I5	12345

L (Logical) Editing

Form:

L*w*

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field, including blanks.

The **L** edit descriptor directs editing between logical values in internal form and their character representations. The **L** edit descriptor must correspond to an input/output list item of type logical.

The input field consists of optional blanks, followed by an optional decimal point, followed by a `T` for true or an `F` for false. Any characters following the `T` or `F` are accepted on input but are not acted upon; therefore, the strings `.TRUE.` and `.FALSE.` are acceptable input forms.

The output field consists of `T` or `F` preceded by *w*–1 blanks.

Examples of L Editing on Input

Input	Format	Value
t	L4	true
.false.	L7	false

Examples of L Editing on Output

Value	Format	Output
true	L4	bbbT
false	L1	F

P (Scale Factor) Editing

Form:

***k*P**

where:

k is the scale factor, an optionally signed integer constant representing a power of ten.

The scale factor, *k*, applies to all subsequently processed **F**, **E**, **D**, **G**, and **Q** edit descriptors until another scale factor is encountered or until format control terminates. The value of *k* is zero at the beginning of each input/output statement.

On input, when an input field using an **F**, **E**, **D**, **G**, or **Q** edit descriptor contains an exponent, the scale factor is ignored. Otherwise, the internal value equals the external value multiplied by $10^{(-k)}$.

On output:

- In **F** editing, the external value equals the internal value multiplied by 10^k.
- In **E**, **D**, and **Q** editing, the external decimal field is multiplied by 10^k. The exponent is then reduced by *k*.
- In **G** editing, fields are not affected by the scale factor unless they are outside the range that can use **F** editing. If the use of **E** editing is required, the scale factor has the same effect as with **E** output editing.

Examples of P Editing on Input

Input	Format	Value
98.765	3P,F8.6	0.098765E20
98.765	-3P,F8.6	98765.
.98765E+2	3P,F10.5	.98765E+2

Examples of P Editing on Output

Value	Format	Output
12.34	2P,F7.2	1234.00
12.34	-2P,F6.4	0.1234
12.34	2P,E10.3	b12.34E+00

S, SP, and SS (Sign Control) Editing

Forms:

S

SP

SS

The **S**, **SP**, and **SS** edit descriptors control the output of plus signs by all subsequently processed **I**, **F**, **E**, **D**, **G**, and **Q** edit descriptors until another **S**, **SP**, or **SS** edit descriptor is encountered or until format control terminates.

S and **SS** specify that plus signs are not to be written. (They produce identical results.) **SP** specifies that plus signs are to be written.

T, TL, TR, and X (Positional) Editing

Forms:

T*c*

TL*c*

TR*c*

*n***X**

where:

c is an unsigned, nonzero, integer constant.

n is an unsigned, nonzero, integer constant.

The **T**, **TL**, **TR**, and **X** edit descriptors specify the position where the transfer of the next character to or from a record starts. This position is:

- For **T***c*, the *c*th character position.
- For **TL***c*, *c* characters backward from the current position. If the value of *c* is greater than or equal to the current position, the next character accessed is position one of the record.
- For **TR***c*, *c* characters forward from the current position.
- For *n***X**, *n* characters forward from the current position.

The **TR** and **X** edit descriptors give identical results.

On input, a **TR** or **X** edit descriptor can specify a position beyond the last character of the record if no characters are transferred from that position.

On output, a **T**, **TL**, **TR**, or **X** edit descriptor does not by itself cause characters to be transferred. If characters are transferred to positions at or after the position specified by the edit descriptor, positions skipped and previously unfilled are filled with blanks. The result is the same as if the entire record were initially filled with blanks.

On output, a **T**, **TL**, **TR**, or **X** edit descriptor can result in repositioning so that subsequent editing with other edit descriptors causes character replacement.

Examples of T, TL, and X Editing on Input

```
150    FORMAT(I4,T30,I4)
200    FORMAT(F6.2,5X,5(I4,TL4))
```

Examples of T, TL, TR, and X Editing on Output

```
50     FORMAT('Column 1',5X,'Column 14',TR2,'Column 25')
100    FORMAT('aaaaa',TL2,'bbbbb',5X,'ccccc',T10,'ddddd')
```

Z (Hexadecimal) Editing

Form:

Z*w*

Z*w.m*

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field, including blanks.

m is an unsigned integer constant that specifies the minimum number of hexadecimal digits to be written. *m* must have a value that is less than or equal to *w*. *m* is useful on output only; it has no effect on input.

The **Z** edit descriptor directs editing between values of any type in internal form and their hexadecimal representation. (A hexadecimal digit is one of 0–9 or A–F.)

On input, *w* hexadecimal digits are edited and form the internal representation for the value of the input list item. The hexadecimal digits in the input field correspond to the rightmost hexadecimal digits of the internal representation of the value assigned to the input list item. *m* has no effect on input.

The output field contains *w* hexadecimal digits, including leading zeros. The digits in the output field correspond to the rightmost *w* hexadecimal digits of the internal representation. If *m* is present, the output field consists of at least *m* hexadecimal digits, and is zero filled on the left, if necessary, until there are *m* hexadecimal digits.

The editing of character data for input or output does not imply blank padding as it does for **A** editing.

Examples of Z Editing on Input

Input	Format	Value
0C	Z2	12
7FFF	Z4	32767

Examples of Z Editing on Output

Value	Format	Output
12	Z4	000C
–1	Z8	FFFFFFFF

O (Octal) Editing

Form:

O*w*

O*w.m*

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field, including blanks.

m is an unsigned integer constant that specifies the minimum number of octal digits to be written. *m* must have a value that is less than or equal to *w*. *m* is useful on output only; it has no effect on input.

The **O** edit descriptor directs editing between values of any type in internal form and their octal representation. (An octal digit is one of 0–7)

On input, *w* octal digits are edited and form the internal representation for the value of the input list item. The octal digits in the input field correspond to the rightmost octal digits of the internal representation of the value assigned to the input list item. *m* has no effect on input.

The output field contains *w* octal digits, including leading zeros. The digits in the output field correspond to the rightmost *w* octal digits of the internal representation. If *m* is present, the output field consists of at least *m* octal digits, and is zero filled on the left, if necessary, until there are *m* octal digits.

The editing of character data for input or output does not imply blank padding as it does for **A** editing.

Examples of O Editing on Input

Input	Format	Value
123	O3	83
120	O3	80

Examples of O Editing on Output

Value	Format	Output
83	O3	123
80	O5	00120

B (Binary) Editing

Form:

B*w*

B*w.m*

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field, including blanks.

m is an unsigned integer constant that specifies the minimum number of binary digits to be written. *m* must have a value that is less than or equal to *w*. *m* is useful on output only; it has no effect on input.

The **B** edit descriptor directs editing between values of any type in internal form and their binary representation. (A binary digit is one of 0 or 1.)

On input, *w* binary digits are edited and form the internal representation for the value of the input list item. The binary digits in the input field correspond to the rightmost binary digits of the internal representation of the value assigned to the input list item. *m* has no effect on input.

The output field contains *w* binary digits, including leading zeros. The digits in the output field correspond to the rightmost *w* binary digits of the internal representation. If *m* is present, the output field consists of at least *m* binary digits, and is zero filled on the left, if necessary, until there are *m* binary digits.

The editing of character data for input or output does not imply blank padding as it does for **A** editing.

Examples of B Editing on Input

Input	Format	Value
111	B3	7
110	B3	6

Examples of B Editing on Output

Value	Format	Output
7	B3	111
6	B5	00110

List–Directed Formatting

In list–directed formatting, editing is controlled by the types and lengths of the data being read or written. An asterisk format identifier specifies list–directed formatting. For example:

```
REAL*8 TOTAL1, TOTAL2
WRITE(6,*) TOTAL1, TOTAL2
```

Note: List–directed formatting can only be used with sequential files.

The characters in a formatted record processed under list–directed formatting constitute a sequence of values separated by value separators:

- A value has the form of a constant or null value.
- A value separator is a comma, slash, or blank. A comma or slash can be preceded and followed by one or more blanks. Blanks in list–directed input records are significant.

List–Directed Input

Input list items in a list–directed **READ** statement are defined by corresponding values in records. The form of each input value must be acceptable for the type of the input list item. An input value is any of the following:

- A value having the form of:
  - *constant*
  - *r*constant*, where *r* is an unsigned, nonzero, integer constant. This form is equivalent to *r* successive appearances of the constant.

constant is an integer, real, double precision, complex, logical, or character constant.

Note: A / in a character constant is not recognized as an escape character.

- A null value, represented by:
  - Two successive commas, with zero or more intervening blanks
  - A comma followed by a slash, with zero or more intervening blanks
  - An initial comma in the record, preceded by zero or more blanks.

More than one null value can be represented by the form *r**, where *r* is an unsigned integer constant. This form is equivalent to *r* successive null values.

A character value can be continued in as many records as required.

The end of a record:

- Has the same effect as a blank separator unless the blank is within a character constant or complex constant.
- Does not cause insertion of a blank or any other character in a character value
- Must not separate two apostrophes representing an apostrophe.

Two or more consecutive blanks are treated as a single blank unless the blanks are within a character value.

A null value has no effect on the definition status of the corresponding input list item.

A slash marks the end of the input list, and list–directed formatting is terminated. If additional items remain in the input list when a slash is encountered, it is as if null values had been specified for those items.

List–Directed Output

List–directed **WRITE** and **PRINT** statements produce values in the order they appear in an output list. Values are written in a form that is valid for the data type of each output list item.

Logical values are written as `T` for the value true and `F` for the value false.

Real values are written as in E or F format editing. (See "E Editing" on page 125 or "F Editing" on page 126 for more information.)

Character values are written as if the **A** edit descriptor were in effect. Character values written with list–directed output formatting cannot be read with list–directed input formatting because apostrophes are not written. You can write out a character value with apostrophes to allow a FORTRAN program to read it in with list–directed formatting.

Slashes, as value separators, and null values are not written.

Arrays are written in column–major order.

The following table shows the width of the written field for any data type and length. The size of the record will be the sum of the field widths plus a byte to separate each non–character field.

Data Type	Length (bytes)	Maximum Field Width (characters)
integer	1 2 4	4 6 11
real	4 8 16	17 25 25
complex	8 16 32	37 53 53
character	n	n
logical	1 2 4	1 1 1

NAMELIST Formatting

NAMELIST Input Data

Input data must be in a special form to be read using a **NAMELIST** list. The first character in each record to be read must be blank. The second character in the first record of a group of data records must be an ampersand (&) immediately followed by the **NAMELIST** name. The **NAMELIST** name must be followed by a blank and must not contain any embedded blanks. This name is followed by data items separated by commas (a comma after the last item is optional). The end of a data group is signaled by `&END`.

Note: **NAMELIST** formatting can only be used with sequential files.

The form of the data items in an input record is:

- Name = Constant
  - The name can be an array element name or a variable name.
  - The constant can be integer, real, complex, logical or character. (Logical constants are: `T`, `.TRUE.`, `F`, `.FALSE.`. If the constants are characters, they must be included between apostrophes.) Character constants can be included between double quotation marks.
  - Subscripts must be integer constants.
- Array Name = Set of Constants (separated by commas)
  - The set of constants consists of the type integer, real, complex, logical or character.
  - The number of constants must be less than or equal to the number of elements in the array.
  - Successive occurrences of the same constant can be represented in the form *c*constant*, where *c* is a nonzero integer constant specifying the number of times the constant is to occur.

The variable names and array names specified in the input file must appear in the **NAMELIST** list, but the order is not significant. A name that has been made equivalent to a name in the input data cannot be substituted for that name in the **NAMELIST** list. The list can contain names of items in **COMMON** but must not contain dummy argument names.

Each data record must begin with a blank followed by a complete variable or array name or constant. You cannot have embedded blanks in names or constants. Trailing blanks after integers and exponents are treated as zeros.

NAMELIST Output Data

When output data is written using a **NAMELIST** list, it is written in a form that can be read using a **NAMELIST** list. All variable and array names specified in the **NAMELIST** list and their values are written out, each according to its type. Character data is included between apostrophes. The fields for the data are made large enough to contain all the significant digits. (See the table on page 135.) The values of a complete array are written out in column–major order.

Chapter 13. Debug Lines

This chapter describes debug lines.

Debug Lines

You can code the letter D in column 1 of any source program line (fixed–form or free–form input format) to indicate that the line is debugging code. The handling of debug lines depends on the **DLINES** compiler option. If you do not specify **DLINES**, the compiler handles such lines as comment lines. If you do specify the **DLINES** option, the compiler interprets the D in column 1 as a blank, and handles such lines as lines of source code.

The initial line of a debugging statement can contain a statement label in columns 2 through 5 for fixed–form source, or as the first nonblank characters (digits) following the D for free–form source.

If you continue a fixed–form debugging statement onto more than one line, every continuation line must have a D in column 1 and a continuation indicator. If the initial line is not a debugging line, you can designate any continuation lines as debug lines provided that the statement is syntactically correct whether or not you specify the **DLINES** option.

If you continue a free–form debugging statement onto more than one line, only the first line can have a D in column 1 and any continuation lines are interpreted as debug lines.

Examples of Debug Lines

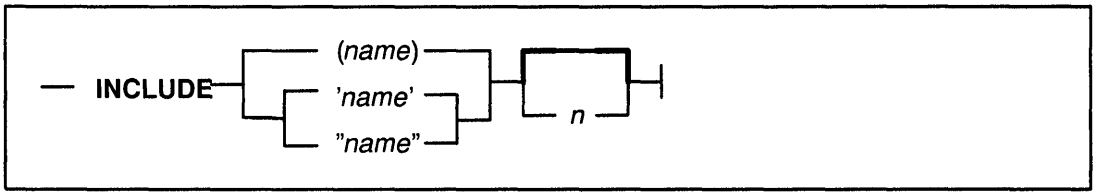
```
D 100 WRITE(6,*) IERROR
D      IF (I.EQ.IDEBUG.AND.
D     +    J.EQ.IDEBUG)      WRITE(6,*) IERROR
       IF (I.EQ.0
D     +           + IDEBUG
      +                   ) WRITE(6,*) INFO
```

Chapter 14. Compiler Directives

This chapter describes the compiler directives:

- **INCLUDE**
- **EJECT**
- **@PROCESS.**

INCLUDE



name
is the name of a group of one or more FORTRAN source statements the compiler is to insert into the current source program. Under AIX, *name* need not specify the full path of the desired file, but it must specify the file extension, if any. The file name should contain only characters allowable in the character set of XL FORTRAN. (See "Characters" on page 7 for the XL FORTRAN character set.)

n
is the value the compiler uses to decide whether to include the file during compilation. It can be any number from 1 through 255. If you specify *n*, the compiler includes the file only if the number appears as a suboption in the **CI** (conditional include) compiler option. If you do not specify *n*, the compiler always includes the file.

The **INCLUDE** compiler directive inserts a specified statement or a group of statements into a program unit. This is a simple insertion function. The compiler does not perform any replacement or editing.

A feature called conditional **INCLUDE** provides a means for selectively activating **INCLUDE** compiler directives within the FORTRAN source during compilation. You specify the included files by means of the **CI** compiler option.

An **INCLUDE** compiler directive can appear anywhere in a FORTRAN source file before the **END** statement, except in situations where the language requires an executable statement. (For example, as the trailer of a logical **IF** statement.) Multiple **INCLUDE** compiler directives can appear in the original source program. No other statement can have an initial line that appears to be an **INCLUDE** compiler directive.

You must not continue an **INCLUDE** compiler directive. Also, you cannot continue the first noncomment line of the included file.

An included file can contain any complete FORTRAN source statements, including other **INCLUDE** compiler directives. XL FORTRAN does not allow recursive **INCLUDE** compiler directives. An **END** statement can be part of the included group. The FORTRAN statements in your included group must be in the same form as the compiling source program (fixed–form or free–form). After the inclusion of all groups, the resulting FORTRAN program must follow all of the FORTRAN rules for sequencing of statements.

XL FORTRAN treats an **INCLUDE** compiler directive with the left and right parenthesis syntax like any other FORTRAN statement. The compiler folds the file name to lowercase unless the **MIXED** option is on. XL FORTRAN ignores blanks in the file name. The file name should not have imbedded left and right parentheses.

For **INCLUDE** compiler directives with single or double quotation marks, all rules for string literals apply:

- The compiler does not fold characters within single or double quotation marks.
- It does not ignore blanks.
- Backslash escape characters apply.

The AIX file system locates the file specified by *name* as follows:

- If the first nonblank character of *name* is `/`, name specifies an absolute file name.
- If the first nonblank character is not `/`, the AIX operating system searches directories in order of decreasing priority:
  - If you specify the `-I`*path* compiler option, the file *path*`/`*name* is searched for.
  - If file *path*`/`*name* is not found:
    - AIX searches the current directory for file *name*.
    - AIX searches the resident directory of the compiling source file for file *name*.
    - AIX searches directory `/usr/include` for file *name*.

Examples of the INCLUDE Compiler Directive

```
INCLUDE '/u/userid/dc101'      ! full absolute file name specified
INCLUDE '/u/userid/dc102.inc'  ! INCLUDE file name has an extension
INCLUDE 'userid/dc103'         ! relative path name specified
INCLUDE (ABCdef)               ! includes file abcdef
INCLUDE 'abcDEF'               ! includes file abcDEF
INCLUDE 'ABCdef'               ! includes file ABCdef
```

EJECT

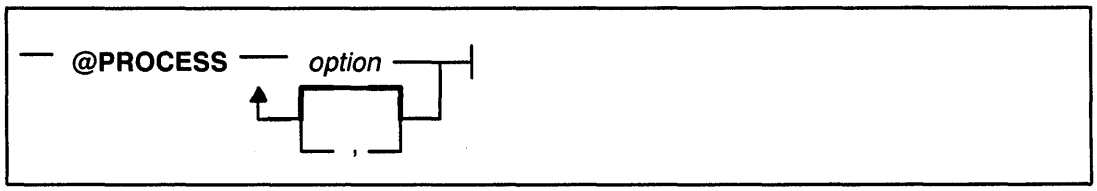
EJECT is a compiler directive. It directs the compiler to start a new full page of the source listing. If no source listing has been requested, this statement is ignored.

An **EJECT** compiler directive can appear anywhere in a FORTRAN source file prior to the **END** statement.

You must not continue an **EJECT** compiler directive. No other statement can have an initial line that appears to be an **EJECT** compiler directive.

If a statement label is on the **EJECT** compiler directive, the compiler discards it. Therefore, you must not reference any label on an **EJECT** compiler directive within the program unit.

@PROCESS



You can use the **@PROCESS** compiler directive in your source file to modify the options specified on the command line, or to change the default setting temporarily if no command line options are in effect.

See the *User's Guide for IBM AIX XL FORTRAN Compiler/6000*, SC09–1257 for details of this compiler directive.

Appendix A. Intrinsic Functions

IBM AIX XL FORTRAN Run Time Environment/6000 supplies a number of standard procedures called intrinsic functions. This appendix describes the intrinsic functions in XL FORTRAN.

You can reference some intrinsic functions by a generic name, some by a specific name, and some by both. A generic name does not require a specific argument type and usually produces a result of the same type as that of the argument, with the exception of type conversion, nearest integer, and absolute value with a complex argument. Generic names simplify the referencing of intrinsic functions, because the same function name can be used with more than one type of argument. The compiler determines the specific function to be used based on the type of the argument specified.

A specific name requires a specific argument type and produces a result of a specific type. Only a specific intrinsic function name can be used as an actual argument when the argument is an intrinsic function. If the actual argument is an intrinsic function for which **INTEGER*2** and **INTEGER*4** arguments are allowed, then the **INTEGER*4** intrinsic function is passed to the subprogram. All references to a dummy procedure that is associated with such an intrinsic function (through argument association) must be references with **INTEGER*4** arguments.

For those intrinsic functions that have more than one argument, all arguments must be of the same type. If the specific or generic name of an intrinsic function appears in the dummy argument list of a function, subroutine, or entry statement in a subprogram, XL FORTRAN does not invoke the intrinsic function. The compiler treats the name as a dummy argument only. You can specify the data type identified with the symbolic name in the same manner as variables and arrays.

A name in an **INTRINSIC** statement must be the specific or generic name of an intrinsic function.

Note that XL FORTRAN provides some **Q** (extended precision) functions. If you specify a **Q** function in a source program, it will actually result in a call to the corresponding **D** (double precision) function.

Referencing an Intrinsic Function

You reference an intrinsic function by using it as a primary in an expression. The result when you invoke an intrinsic function depends on the values of the actual arguments. The resulting value is available to the expression that contains the function reference. (See "Function Reference" on page 83 for the discussion on how to invoke functions.)

The actual arguments that constitute the argument list must agree in order, number, and type with those required by the definition of the intrinsic function. They can be any expression of the specified type. An actual argument in an intrinsic function reference can be any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant. Actual arguments cannot be names of arrays or procedures.

You can use the specific name of an intrinsic function that appears in an **INTRINSIC** statement as an actual argument in an external procedure reference. Do not use the names of intrinsic functions as actual arguments for type conversion or for choosing the largest and smallest value.

Intrinsic Function Rules and Restrictions

- An **IMPLICIT** statement does not change the type of an intrinsic function.
- Mathematically undefined arguments or arguments that exceed the numeric range of the processor cause undefined function results.
- If an intrinsic name appears in an **INTRINSIC** statement, XL FORTRAN assumes that you really want this name as an intrinsic function regardless of where else the name can be used.

Definition problems arise when an intrinsic name does not appear in an **INTRINSIC** statement.

- You cannot use the intrinsic name (specific or generic) as an intrinsic function if you have used it previously in one of the following ways:
  - In any specification statement other than an explicit type or **INTRINSIC** statement (including statement function definitions and dummy arguments).
  - If you dimension it or it has initial data in an explicit type statement.
  - As the name of a **COMMON** block.
- If you define a specific name as in the previous item, you can still use the generic name (if different) for a function reference.
- If the first usage is as a variable, the name becomes fixed and you can no longer use it as a function. This restriction applies to both specific and generic names.
- If a specific name becomes fixed in this manner, you can still use the generic name (if different) as an intrinsic function. If the generic name becomes fixed, you can still use the specific names associated with the generic as functions.
- If the first usage is as a function, XL FORTRAN treats the name as a function and you can no longer use it as a variable.

Intrinsic Functions

The following table lists the the intrinsic function, the equation that represents the function definition,its generic name (if there is one), its specific name, the number of arguments and the type of argument, and the type of the function value returned by the intrinsic function.

Intrinsic Function	Definition (See Notes below)	Generic Name	Specific Name	Arg. No. & Type	Argument Range	Func. Type
Conversion to type integer	y=(sign of x)*n where n is the largest integer $\leq \|x\|$	INT	— —	1 I*4 1 I*2	Any integer argument	I*4 I*4
			INT IFIX IDINT IQINT	1 R*4 1 R*4 1 R*8 1 R*8	Any real argument	I*4 I*4 I*4 I*4
	For $z=x_1+x_2i$, $y=INT(x_1)$	INT	— —	1 X*8 1 X*16	Any complex argument	I*4 I*4
Conversion to type integer	y=(sign of x)*n where n is the largest integer $\leq \|x\|$		HFIX	1 R*4	Any real argument	I*2

Intrinsic Function	Definition (See Notes below)	Generic Name	Specific Name	Arg. No. & Type	Argument Range	Func. Type
Conversion to type real		REAL	REAL FLOAT FLOAT	1 I*4 1 I*4 1 I*2	Any integer argument	R*4 R*4 R*4
			— SNGL SNGLQ	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*4 R*4
	For $z=x_1+x_2i$, $y=REAL(x_1)$	REAL	—	1 X*8	Any complex argument	R*4
Conversion to type double precision		REAL	DREAL QREAL	1 X*16 1 X*16	Any complex argument	R*8 R*8
Conversion to type double precision		DBLE	DFLOAT DFLOAT	1 I*4 1 I*2	Any integer argument	R*8 R*8
			DBLE DBLEQ	1 R*4 1 R*8	Any real argument	R*8 R*8
	For $z=x_1+x_2i$, $y=DBLE(x_1)$	DBLE	— —	1 X*8 1 X*16	Any complex argument	R*8 R*8
Conversion to type double precision		QEXT	QFLOAT QFLOAT	1 I*4 1 I*2	Any integer argument	R*8 R*8
			QEXT QEXTD	1 R*4 1 R*8	Any real argument	R*8 R*8
Conversion to type complex	$y=x_1+x_2i$ where $x_1=REAL(arg_1)$ and $x_2=REAL(arg_2)$	CMPLX	— —	1/2 I*4 1/2 I*2	Any integer argument	X*8 X*8
			CMPLX —	1/2 R*4 1/2 R*8	Any real argument	X*8 X*8
			— —	1 X*8 1 X*16	Any complex argument	X*8 X*8
Conversion to type double complex	$y=x_1+x_2i$ where $x_1=REAL(arg_1)$ and $x_2=REAL(arg_2)$	DCMPLX	— —	1/2 I*4 1/2 I*2	Any integer argument	X*16 X*16
			DCMPLX —	1/2 R*4 1/2 R*8	Any real argument	X*16 X*16
			— —	1 X*8 1 X*16	Any complex argument	X*16 X*16
Conversion to type double complex	$y=x_1+x_2i$ where $x_1=REAL(arg_1)$ and $x_2=REAL(arg_2)$ ($y=x_1+0i$ if only x_1 is present)	QCMPLX	QCMPLX	1/2 R*8	Any real argument	X*16

Intrinsic Function	Definition (See Notes below)	Generic Name	Specific Name	Arg. No. & Type	Argument Range	Func. Type
Truncation	y=(sign of x)*n where n is the largest integer $\leq\|x\|$	AINT	AINT DINT QINT	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*8 R*8
Nearest whole number	If $x\geq 0$, y=(sign of x)*n where n is the largest integer $\leq\|x+.5\|$ If $x<0$, y=(sign of x)*n where n is the largest integer $\leq\|x-.5\|$	ANINT	ANINT DNINT	1 R*4 1 R*8	Any real argument	R*4 R*8
Nearest integer	If $x\geq 0$, y=(sign of x)*n where n is the largest integer $\leq\|x+.5\|$ If $x<0$, y=(sign of x)*n where n is the largest integer $\leq\|x-.5\|$	NINT	NINT IDNINT	1 R*4 1 R*8	Any real argument	I*4 I*4
Absolute value	y=|x|	ABS	IABS IABS	1 I*4 1 I*2	Any integer argument	I*4 I*2
			ABS DABS QABS	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*8 R*8
	$y=\|z\|=(x_1^2+x_2^2)^{<1/2>}$	ABS	CABS CDABS CQABS ZABS	1 X*8 1 X*16 1 X*16 1 X*16	Any complex argument	R*4 R*8 R*8 R*8
Remainder-ing (modular arithmetic)	y=(sign of x1*x2) $*(x_1-n*x_2)$ where n is the largest integer $\leq\|x1/x2\|$	MOD	MOD MOD AMOD DMOD QMOD	2 I*4 2 I*2 2 R*4 2 R*8 2 R*8	$x_2\neq 0$	I*4 I*2 R*4 R*8 R*8
Transfer of sign	If $x_2\geq 0$, $y=\|x_1\|$ If $x_2<0$, $y=-\|x_1\|$	SIGN	ISIGN ISIGN	2 I*4 2 I*2	Any integer argument	I*4 I*2
			SIGN DSIGN QSIGN	2 R*4 2 R*8 2 R*8	Any real argument	R*4 R*8 R*8
Positive difference	If $x_1>x_2$, $y=x_1-x_2$ If $x_1\leq x_2$, y=0	DIM	IDIM IDIM	2 I*4 2 I*2	Any integer argument	I*4 I*2
			DIM DDIM QDIM	2 R*4 2 R*8 2 R*8	Any real argument	R*4 R*8 R*8
Double precision product	$y=x_1*x_2$		DPROD	2 R*4	Any real argument	R*8
Choosing largest value	$y=\max(x_1,...x_n)$	MAX	MAX0 MAX0	≥2 I*4 ≥2 I*2	Any integer argument	I*4 I*2
			AMAX1 DMAX1 QMAX1	≥2 R*4 ≥2 R*8 ≥2 R*8	Any real argument	R*4 R*8 R*8

Intrinsic Function	Definition (See Notes below)	Generic Name	Specific Name	Arg. No. & Type	Argument Range	Func. Type
Choosing largest value	$y=max(x_1,...x_n)$		AMAX0 AMAX0	≥2 I*4 ≥2 I*2	Any integer argument	R*4 R*4
			MAX1 MAX1	≥2 R*4 ≥2 R*8	Any real argument	I*4 I*4
Choosing smallest value	$y=min(x_1,...x_n)$	MIN	MIN0 MIN0	≥2 I*4 ≥2 I*2	Any integer argument	I*4 I*2
			AMIN1 DMIN1 QMIN1	≥2 R*4 ≥2 R*8 ≥2 R*8	Any real argument	R*4 R*8 R*8
Choosing smallest value	$y=min(x_1,...x_n)$		AMIN0 AMIN0	≥2 I*4 ≥2 I*2	Any integer argument	R*4 R*4
			MIN1 MIN1	≥2 R*4 ≥2 R*8	Any real argument	I*4 I*4
Imaginary part of a complex		IMAG	AIMAG DIMAG QIMAG	1 X*8 1 X*16 1 X*16	Any complex argument	R*4 R*8 R*8
Complex conjugate	$y=x_1-x_2i$ for argument x_1+x_2i	CONJG	CONJG DCONJG QCONJG	1 X*8 1 X*16 1 X*16	Any complex argument	X*8 X*16 X*16
Square root	$y=x^{(1/2)}$	SQRT	SQRT DSQRT QSQRT	1 R*4 1 R*8 1 R*8	$x\geq 0$	R*4 R*8 R*8
	$y=z^{(1/2)}$	SQRT	CSQRT CDSQRT CQSQRT ZSQRT	1 X*8 1 X*16 1 X*16 1 X*16	Any complex argument	X*8 X*16 X*16 X*16
Exponential	$y=e^x$	EXP	EXP	1 R*4	$x\leq 88.7228$	R*4
			DEXP QEXP	1 R*8 1 R*8	$x\leq 709.7828$	R*8 R*8
	$y=e^z$	EXP	CEXP	1 X*8	$x_1\leq 88.7228$, x_2 is any real argument	X*8
			CDEXP CQEXP ZEXP	1 X*16 1 X*16 1 X*16	$x_1\leq 709.7828$, x_2 is any real argument	X*16 X*16 X*16
Natural logarithm	$y=log_e(x)$ or y=ln x	LOG	ALOG DLOG QLOG	1 R*4 1 R*8 1 R*8	x>0	R*4 R*8 R*8
	$y=log_e(z)$	LOG	CLOG CDLOG CQLOG ZLOG	1 X*8 1 X*16 1 X*16 1 X*16	$z\neq 0+0i$	X*8 X*16 X*16 X*16
Common logarithm	$y=log_{10}x$	LOG10	ALOG10 DLOG10 QLOG10	1 R*4 1 R*8 1 R*8	x>0	R*4 R*8 R*8

Intrinsic Function	Definition (See Notes below)	Generic Name	Specific Name	Arg. No. & Type	Argument Range	Func. Type
Sine	y=sin(x)	SIN	SIN DSIN QSIN	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*8 R*8
	y=sin(z)	SIN	CSIN	1 X*8	$\|x_2\| \leq 88.7228$, x_1 is any real argument	X*8
			CDSIN CQSIN ZSIN	1 X*16 1 X*16 1 X*16	$\|x_2\| \leq 709.7828$, x_1 is any real argument	X*16 X*16 X*16
Cosine	y=cos(x)	COS	COS DCOS QCOS	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*8 R*8
	y=cos(z)	COS	CCOS	1 X*8	$\|x_2\| \leq 88.7228$, x_1 is any real argument	X*8
			CDCOS CQCOS ZCOS	1 X*16 1 X*16 1 X*16	$\|x_2\| \leq 709.7828$, x_1 is any real argument	X*16 X*16 X*16
Tangent	y=tan(x)	TAN	TAN DTAN QTAN	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*8 R*8
Arcsine	y=arcsin(x), $-\pi/2 \leq y \leq \pi/2$	ASIN	ASIN DASIN QARSIN	1 R*4 1 R*8 1 R*8	$\|x\| \leq 1$	R*4 R*8 R*8
Arccosine	y=arccos(x), $0 \leq y \leq \pi$	ACOS	ACOS DACOS QARCOS	1 R*4 1 R*8 1 R*8	$\|x\| \leq 1$	R*4 R*8 R*8
Arctangent	y=arctan(x) $-\pi/2 \leq y \leq \pi/2$	ATAN	ATAN DATAN QATAN	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*8 R*8
Arctangent	y= arctan(x_1/x_2) $-\pi \leq y \leq \pi$	ATAN2	ATAN2 DATAN2 QATAN2	2 R*4 2 R*8 2 R*8	$x_1 \neq 0$ and $x_2 \neq 0$	R*4 R*8 R*8
Hyperbolic sine	$y=(e^x-e^{-x})/2$	SINH	SINH	1 R*4	$\|x\| \leq 89.4159$	R*4
			DSINH QSINH	1 R*8 1 R*8	$\|x\| \leq 709.7828$	R*8 R*8
Hyperbolic cosine	$y=(e^x+e^{-x})/2$	COSH	COSH	1 R*4	$\|x\| \leq 89.4159$	R*4
			DCOSH QCOSH	1 R*8 1 R*8	$\|x\| \leq 709.7828$	R*8 R*8
Hyperbolic tangent	$y=(e^x-e^{-x})/(e^x+e^{-x})$	TANH	TANH DTANH QTANH	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*8 R*8
Conversion to type integer	Position of x in collating sequence		ICHAR	1 C*1		I*4

Intrinsic Function	Definition (See Notes below)	Generic Name	Specific Name	Arg. No. & Type	Argument Range	Func. Type
Conversion to type character	Character corresponding to position of x in collating sequence		CHAR CHAR	1 I*4 1 I*2		C*1 C*1
Length	Length of x		LEN	1 C		I*4
Index of a substring	Location of substring x_2 in string x_1		INDEX	2 C		I*4
Alphanumeric greater than or equal	$x_1 \geq x_2$ Comparison is by collating sequence		LGE	2 C		L*4
Alphanumeric greater than	$x_1 > x_2$ Comparison is by collating sequence		LGT	2 C		L*4
Alphanumeric less than or equal	$x_1 \leq x_2$ Comparison is by collating sequence		LLE	2 C		L*4
Alphanumeric less than	$x_1 < x_2$ Comparison is by collating sequence		LLT	2 C		L*4
Inclusive or	$y=or(x_1,x_2)$		IOR IOR OR OR	2 I*4 2 I*2 2 I*4 2 I*2	Any integer argument	I*4 I*2 I*4 I*2
Logical and	$y=and(x_1,x_2)$		IAND IAND AND AND	2 I*4 2 I*2 2 I*4 2 I*2	Any integer argument	I*4 I*2 I*4 I*2
Logical complement	$y=not(x_1)$		NOT NOT	1 I*4 1 I*2	Any integer argument	I*4 I*2
Exclusive or	$y=xor(x_1,x_2)$		IEOR IEOR XOR XOR	2 I*4 2 I*2 2 I*4 2 I*2	Any integer argument	I*4 I*2 I*4 I*2
Shift operations	$y=ishft(x_1,x_2)$ x_1 is shifted x_2 bits right if $x_2<0$ or to left if $x_2>0$		ISHFT ISHFT	2 I*4 2 I*2	$\|x_2\| \leq 32$	I*4 I*2
	$y=lshft(x_1,x_2)$ x_1 is shifted x_2 bits left		LSHIFT LSHIFT	2 I*4 2 I*2	$\|x_2\| \leq 32$	I*4 I*2
	$y=rshft(x_1,x_2)$ x_1 is shifted x_2 bits right		RSHIFT RSHIFT	2 I*4 2 I*2	$\|x_2\| \leq 32$	I*4 I*2
Shift operations	$y=ishftc(x_1,x_2,x_3)$ Rightmost x_2 bits of x_1 shifted circularly by x_3 bits		ISHFTC ISHFTC	3 I*4 3 I*2	$\|x_2\| \leq 32$ $\|x_3\| \leq 32$	I*4 I*2

Intrinsic Function	Definition (See Notes below)	Generic Name	Specific Name	Arg. No. & Type	Argument Range	Func. Type
Bit test	y=btest(x_1,x_2) y=true if bit x_2 of x_1=1 or false if bit x_2 of x_1=0		BTEST BTEST	3 I*4 3 I*2	$0 \le x_2 \le 31$	L*4 L*4
Bit set	y=bitset(x_1,x_2) sets bit x_2 of x_1 to 1		IBSET IBSET	2 I*4 2 I*2	$0 \le x_2 \le 31$	I*4 I*2
Bit set	y=bitclear(x_1,x_2) sets bit x_2 of x_1 to 0		IBCLR IBCLR	2 I*4 2 I*2	$0 \le x_2 \le 31$	I*4 I*2
Bit setting	Extracts a subfield of x_3 bits from x_1, starting with bit position x_2 and extending left for x_3 bits		IBITS IBITS	3 I*4 3 I*2	$0 \le x_2 \le 31$ $0 \le x_3 \le 31$	I*4 I*2
Error function	y=erf(x)	ERF	ERF DERF QERF	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*8 R*8
Error function	y=1–erf(x)	ERFC	ERFC DERFC QERFC	1 R*4 1 R*8 1 R*8	Any real argument	R*4 R*8 R*8
Gamma	y=gamma(x)	GAMMA	GAMMA	1 R*4	$0 < x \le 35.0400$	R*4
			DGAMMA	1 R*8	$2.3561D{-}304 \le x \le 2^{1024}$	R*8
Log gamma	y= $\log_e$gamma(x)	LGAMMA	ALGAMA	1 R*4	$0 < x \le 4.0850E36$	R*4
			DLGAMA	1 R*8	$2.3561D{-}304 \le x \le 2^{1024}$	R*8

Figure 7. Table of Intrinsic Functions

Notes About Definitions:

Definitions use familiar mathematical function names, which have familiar mathematical meanings or are defined below.

The bits in the bit–manipulation functions **BTEST**, **IBSET**, and **IBCLR** are numbered from right to left, beginning at zero.

The result of a function of type complex is the principal value.

All the bit manipulation and logical functions allow **INTEGER*2** arguments, giving an **INTEGER*2** result (except for **BTEST**, for which the result is **LOGICAL*4**).

Meanings of symbols:

x denotes a single argument.

x_i denotes the i–th argument when a function accepts more than one argument.

|x| denotes the absolute value of x.

sign(x) is +1 if $x \ge 0$ or –1 if $x < 0$.

y denotes a function result.

z denotes a complex argument.

Appendix B. XL FORTRAN Run Time Environment

This appendix discusses the various library subprograms, and the error handling support that XL FORTRAN provides.

XL FORTRAN Subprograms

The XL FORTRAN library contains the following categories of subprograms to provide various run–time capabilities:

- Input/output operation subprograms
- Data conversion subprograms
- Mathematical, character, and bit subprograms
- Service and utility subroutine subprograms
- Termination subprograms.

Note that the XL FORTRAN compiler provides no extended error handling subprogram support.

Input/output operation subprograms

FORTRAN language input/output statements are not implemented directly in the generated code output by the FORTRAN compiler, but instead are implemented by generated calls to the FORTRAN library. The statements so supported are:

- **READ** and **WRITE** in all their forms:
  - Formatted
  - Unformatted
  - **NAMELIST**
  - List–directed
  - Sequential
  - Direct access.
- **PRINT**
- **BACKSPACE**
- **REWIND**
- **ENDFILE**
- **OPEN**
- **CLOSE**
- **INQUIRE**
- **FORMAT**.

Data conversion subprograms

These subprograms are intrinsic functions, and have been described in Appendix A.

Mathematical, character, and bit subprograms

These subprograms make up the intrinsic functions, which the programmer refers to directly by name, and the notational functions, which are accessed in response to mathematical notation (for example, exponentiation).

Service and utility subprograms
These subprograms (for example, to obtain time accounting information or issue an operating system command from FORTRAN) are called by the programmer directly.

Termination subprograms
The following statements generate calls to these subprograms:

- **STOP**
- **PAUSE**.

Mathematical, Character, and Bit Subprograms

Explicitly Called Subprograms

All explicitly called subprograms are FORTRAN intrinsic functions. They have been described in Appendix A, "Intrinsic Functions".

Implicitly Called Subprograms

The implicitly called subprograms are executed as a result of certain notation appearing in a FORTRAN source statement. The XL FORTRAN compiler generates the instructions necessary to call the appropriate subprogram.

These subprograms are used in the following circumstances:

- Division of complex numbers
- Some forms of exponentiation.

Service and Utility Subprograms

The library provides utility services that are available to the FORTRAN programmer. The subprograms that are subroutines (all but **IARGC**, **MCLOCK**, **IRAND**, and **RAND**) are invoked through use of the **CALL** statement.

The following table lists the available service and utility subprograms, displays the format you should use for invoking them, and gives a description of the function that each performs.

Subprogram Name	Format	Description
ABORT	CALL ABORT( )	The **ABORT** subroutine terminates the program that calls it. It truncates all open files to the current position of the file pointer, closes all open files, and then stops execution.
FPGETS FPSETS	INCLUDE '/usr/include/fpdc.h' CALL FPGETS(FPSTAT) CALL FPSETS(FPSTAT) . . . INCLUDE '/usr/include/fpdc.h' INCLUDE '/usr/include/fpdt.h' END	The subroutines **FPGETS** and **FPSETS** retrieve and set the status of the floating–point operations, respectively. The BLOCK DATA include file fpdc.h contains the data declarations (specification statements) for the two subroutines. The include file fpdt.h contains the data initializations (data statements) and must be included in a block data subprogram. **FPGETS** retrieves the floating–point process status and stores the result in a logical array called fpstat. **FPSETS** sets the floating–point status equal to the logical array fpstat. This array contains logical values that enable or disable system checking for various floating–point errors.
GETARG	INTEGER I1 CHARACTER*N C1 CALL GETARG(I1,C1)	The **GETARG** subroutine returns a command line argument of the current process. I1 is an integer argument that specifies which command line argument to return. C1 is an argument of character type and will contain, upon return from **GETARG**, the command line argument. If I1 is equal to 0, the program name is returned.
GETENV	CHARACTER*N C1 CALL GETENV('env_name',C1)	The **GETENV** subroutine returns the character string value of the specified environment variable contained in the .profile file of the current directory. The environment variable is specified through the first argument, 'env_name' and the character string value of the environment variable is returned in the second argument, C1. If no such environment variable exists, blanks are returned.
IARGC	INTEGER I1 I1 = IARGC( )	The **IARGC** function returns an integer that represents the number of arguments following the program name that have been entered on the command line at run time.
IRAND RAND SRAND	INTEGER I1 REAL R1 CALL SRAND(I1) I1 = IRAND( ) CALL SRAND(R1) R1 = RAND( )	The **IRAND** and **RAND** functions generate uniform random numbers. The **IRAND** function returns a positive integer number greater than 0 and less than or equal to 32768. The **RAND** function returns a positive real number greater than 0 and less than 1.0. The **SRAND** subroutine is used to provide the seed value for the random number generator.
MCLOCK	INTEGER I1 I1 = MCLOCK( )	The **MCLOCK** function returns time accounting information about the current process and its child processes. The returned value is the sum of the current process's user time and the user and system time of all child processes. The unit of measure is one–sixtieth (1/60) of a second.

Subprogram Name	Format	Description
MVBITS	INTEGER I1,I2,I3,I4,I5 CALL MVBITS(I1,I2,I3,I4,I5)	**MVBITS** is a subroutine that moves I_3 bits, starting at position I_2 of argument I_1, to position I_5 of argument I_4.
SIGNAL	INTEGER I1 EXTERNAL INTFNC CALL SIGNAL(I1,INTFNC) INCLUDE 'fexcp.h' CALL SIGNAL(SIGTRAP,xl__trce)	The **SIGNAL** subroutine allows a process to specify a function to be invoked upon receipt of a specific signal. The first argument I1 specifies the signal. The second argument **INTFNC** specifies the user–defined procedure to be invoked upon receipt of the specified signal. This call to signal is used by the exception handler to produce the traceback code. **SIGTRAP** has a value of 5 to indicate it is input for the **tracetrap** instruction. xl__trce is a predefined procedure that generates the traceback code.
SYSTEM	CHARACTER*N C1 CALL SYSTEM(C1)	The **SYSTEM** subroutine allows an operating system command to be issued from FORTRAN. It passes the argument C1 to the operating system as input. The operating system accepts and executes the command specified by this argument as if it were typed from a terminal. The current process pauses until the command is completed and control is returned from the operating system.

Figure 8. FORTRAN Library: Service and Utility Subprograms

Error Handling Support

Compiler Detected Errors

If the XL FORTRAN compiler detects a source error of severity level S, exception code is generated in place of the code for the statement in error. At run time, if this point in the program is reached, the program halts.

The return code at the end of compilation is set to 0 if the highest severity level of all errors diagnosed is E, W, L, or I, or less than *halt_sev* if the **HALT** compiler option has been specified. Otherwise, the return code is set to one of the following values:

1	A severe or unrecoverable error has been detected that is not one of the others listed here.
40	An option error has been detected.
41	A configuration file error has been detected.
250	An "out of memory" error has been detected. The **xlf** command cannot allocate any more memory for its use.
251	A signal received error has been detected. A fatal error or interrupt signal is received.
252	A "file not found" error has been detected.
253	An input/output error has been detected. Cannot read or write files.
254	A fork error has been detected. Cannot create a new process.
255	An execution error has been detected. The error is produced when executing a process.

FORTRAN Exception Handling and Traceback Facilities

The following cases will cause an exception at run time:

1. Subscript out of bounds if you specify the **CHECK** option at compile time.
2. Substring out of bounds if you specify the **CHECK** option at compile time.
3. Fixed point division by zero.
4. The flow of control in the program reaches a location for which a semantic error with severity level of S was issued when the program was compiled.

Using Traceback Facilities

If your program causes a run–time exception, you can obtain a traceback by having your program call **SIGNAL** with *SIGTRAP* and *xl__trce* as parameters at some point before the exception occurs.

For each routine called, the traceback shows the offset number of the routine in which the exception occurred. The offset number can be mapped to the source line number by checking the offset on the listing. (–**qlist** compiler option will produce a listing.)

The **dbx** symbolic debugger can also produce a traceback showing the sequence of routine calls that led to the exception. When you run the program within **dbx** a specific error message with the line number of the instruction causing the exception is written to standard error. Use the **where** or **trace** subcommand in **dbx** to obtain a complete traceback written to standard error showing the source line number and the sequence of instructions leading up to the exception.

Example

The following program produces a run–time exception:

```
PROGRAM TT
CALL A
END

SUBROUTINE A
INCLUDE 'fexcp.h'
DIMENSION A(10)
CALL SIGNAL(SIGTRAP,xl__trce)
I=11
A(I)=0
END
```

If we compile the program above with the **CHECK** option in effect, then traceback table will show the following:

```
offset 34 in procedure A
offset 10 in procedure TT
offset 0 in procedure START
```

Appendix C. XL FORTRAN Compiler/6000 Extensions

The following list of extensions indicates those items that are changes from SAA FORTRAN. Some of the items listed here are implementation differences, and some are actual extensions over SAA.

XL FORTRAN Extensions

Language Syntax

- The XL FORTRAN character set includes the following special characters: ! " % < >
- XL FORTRAN uses the ASCII collating sequence.
- The maximum length of a name is 250 characters by default.
- Both $ and _ are treated as letters when used in a name, and may be used as the first character.
- XL FORTRAN allows columns 1 through 5 to contain characters, but they are ignored.
- A statement can have up to 99 continuation lines.
- A c in column 1 of a fixed–form input line indicates a comment also.
- Inline comments are permitted, and are indicated by !.
- Free–form input format is permitted.
- Tab characters can be used.
- XL FORTRAN includes the following: **DOUBLE COMPLEX**, **STATIC**, **AUTOMATIC**, **NAMELIST**, **DO WHILE**, **END DO**, **INCLUDE**, **EJECT**, and **@PROCESS**.
- **DATA** statements can be interspersed with any specification statements except **IMPLICIT**.

Data Types and Constants

- The data types include **INTEGER*1**, **REAL*16**, **COMPLEX*32**, **DOUBLE COMPLEX**, and **LOGICAL*2**.
- XL FORTRAN also allows **E** or **Q** to specify a **REAL*8** exponent.
- A complex constant can be specified with a pair of real or integer constant expressions enclosed in parentheses and separated by a comma.
- The abbreviations **T** and **F** can be used for **.TRUE.** and **.FALSE.** respectively.
- Character constants can also be delimited by double quotation marks.
- XL FORTRAN recognizes a set of backslash escapes for compatibility with the C language character strings.
- The following constants are included: hexadecimal, octal, binary, and Hollerith.

Variables and Arrays

- If the first character of a variable name is the underscore (_) or the currency symbol ($), the variable is defined implicitly as a real of length 4.
- The maximum number of dimensions that can be declared is 20.

Expressions

- Arithmetic primaries include:
  - Hexadecimal constant
  - Octal constant
  - Binary constant
  - Hollerith constant.
- Character primaries include:
  - Hexadecimal constant
  - Octal constant
  - Binary constant
  - Hollerith constant.
- Logical primaries include:
  - Hexadecimal constant
  - Octal constant
  - Binary constant
  - Hollerith constant.
- The relational operators include the following: < <= == <> > >=
- XL FORTRAN always evaluates a relational expression to a **LOGICAL*4** result, but you can convert the result in an assignment statement to a **LOGICAL*1** or **LOGICAL*2**.
- XL FORTRAN allows the logical operator **.XOR.** for logical nonequivalence.

Arithmetic Assignment Statement

- XL FORTRAN allows the following data types:
  - **INTEGER*1**
  - **REAL*16**
  - **COMPLEX*32, DOUBLE COMPLEX**.

CALL Statement

- Recursion is allowed if the **RECUR** compiler option is specified. Procedures can call themselves, either directly or through a chain of other procedures.
- The argument list keywords **%VAL** and **%REF** aid interlanguage calls by allowing arguments to be passed by value and reference respectively.

COMMON Statement

- If data is misaligned because of its specification in **COMMON**, it can still be processed. Misaligned data can also be passed as arguments to subprograms. Note that any use of misaligned data will adversely affect the performance of the program.

DATA Statement

- XL FORTRAN allows a hexadecimal, octal, binary, or Hollerith constant to initialize a variable, array element or array of any type, or a substring.
- If a binary or octal constant initializes a complex data type, one constant is used that initializes both the real and the imaginary parts. The constant is not enclosed in parentheses. If the constant is smaller than the length (in bytes) of the entire complex entity, zeros are added on the left. If the constant is larger, the leftmost bits are truncated.

Debug Lines

- Debug lines.

Dimension Statement

- Arrays with a maximum of 20 dimensions may be specified.

DO Statement

- The terminal statement of a **DO** loop can be a labeled **END DO** statement.
- If the optional statement label stn is omitted, the terminal statement of the **DO** loop must be an **END DO** statement.

DO WHILE Statement

- **DO WHILE** statement.

EJECT Compiler Directive

- **EJECT** compiler directive.

END DO Statement

- **END DO** statement.

END Statement

- An end–of–line comment, initiated by `!`, can appear on the same line as an **END** statement.

ENTRY Statement

- If the **RECUR** compiler option is specified, a subprogram can reference, either directly or indirectly, any of its **ENTRY** names. If the recursion compiler option is not specified, a subprogram must not refer to itself, or any of its entry points, either directly or indirectly.
- If **IMPLICIT NONE** has been specified in a function subprogram, the **FUNCTION** name and any **ENTRY** names must be explicitly typed.

EQUIVALENCE Statement

- Any missing subscript is assumed to be the lower bound of the corresponding dimension of the array.
- An entity declared as **AUTOMATIC** cannot be used as an argument in an **EQUIVALENCE** statement.

Explicit Type Statement

- Explicit type statements include:
  - **INTEGER*1**
  - **REAL*16**
  - **COMPLEX*32, DOUBLE COMPLEX**
  - **LOGICAL*2**
  - **STATIC**
  - **AUTOMATIC.**
- XL FORTRAN allows a hexadecimal, octal, binary, or Hollerith constant to initialize a variable, array element or array of any type, or a substring.
- If a binary or octal constant initializes a complex data type, one constant is used that initializes both the real and the imaginary parts. The constant is not enclosed in parentheses. If the constant is smaller than the length (in bytes) of the entire complex entity, zeros are added on the left. If the constant is larger, the leftmost bits are truncated.

FORMAT Statement

Code	Format	Description
	"literal"	Literal data (character constant)
Q	aQw.d	Extended precision data fields (handled as double precision by XL FORTRAN.)
G	aGw.dDe	Real, integer, or logical data fields
Z	aZw.m	Hexadecimal data fields
O	aOw.m	Octal data fields
B	aBw.m	Binary data fields
$	$	Suppress end–of–record

- Formatting extreme values

  The "extreme values" used in the IEEE binary floating–point system include +infinity, –infinity and Not–a–Number (NaN). Infinity is the result of floating–point overflow. NaN is produced by certain invalid operations, such as division by zero.

  On the output of nonextreme values, the entire field is filled with asterisks if the field width as specified by the associated format specifier is inadequate to display the value.

  When extreme values are printed in `D`, `E`, `F`, `G` or `Q` format, "`INF`", "`–INF`" and "`NaNQ`" are produced for the internal values +infinity, –infinity and NaN, respectively.
- Commas can be used as value separators in the input record for a formatted read of noncharacter variables. The commas override the field lengths given in a **FORMAT** statement.
- The comma can be omitted between a `P` format code and an immediately following `Q` format code. If the comma is omitted in other circumstances, characters in the format specification are skipped until the next valid format code is found.
- If the form `Zw.m` is used on input, the value of m has no effect. If `.m` is present, the output field consists of at least `m` hexadecimal digits, and is zero–filled on the left, if necessary, until there are m hexadecimal digits.
- `O` format code (octal).

- B format code (binary).
- Character constants can appear in a **FORMAT** statement enclosed in double quotation marks.
- Dollar (end–of–record suppression).

FUNCTION Statement

- **DOUBLE COMPLEX** in a **FUNCTION** statement specifies a double precision complex function.
- If the **IMPLICIT NONE** statement is used within the function subprogram, the type of the function must be explicitly declared in either the **FUNCTION** statement or an explicit type statement.
- Because recursion is allowed, the function name can be referenced within the function subprogram in a context that requires the function to be evaluated. If the function name is used as an actual argument, a variable name and not a procedure name is passed to the subprogram.
- An actual argument can be an argument list keywords (**%VAL** or **%REF**).

Computed GO TO Statement

- The expression to be evaluated in a computed **GO TO** statement can be any arithmetic expression. If it is noninteger, it is converted to an integer value before use.
- A computed **GO TO** can be the terminal statement of a **DO WHILE** loop.

IMPLICIT Type Statement

- The **IMPLICIT NONE** statement.
- **IMPLICIT** type includes:
  - **INTEGER*1**
  - **REAL*16**
  - **COMPLEX*32**
  - **DOUBLE COMPLEX**
  - **LOGICAL*2**
  - **STATIC**
  - **AUTOMATIC**
  - **UNDEFINED**.
- A letter or range of letters that is specified as **STATIC** or **AUTOMATIC** can also appear in an **IMPLICIT** statement for any of the other types, except for **AUTOMATIC** and **STATIC**.
- **IMPLICIT NONE** turns off all implicit data typing defaults. When **IMPLICIT NONE** is specified, the data types of all symbolic names in the program unit must be explicitly declared. The compiler issues a diagnostic message for each symbolic name that is used but does not appear in an explicit type statement.
- When **IMPLICIT NONE** is specified, there must be no other **IMPLICIT** statement in the same program unit (except that **STATIC** and **AUTOMATIC** storage class **IMPLICIT** statements can also be specified). **IMPLICIT NONE** must precede all other specification statements. Explicit type statements must then be used to specify the data types of all entities used in the program unit.

- **IMPLICIT UNDEFINED** turns off the implicit data typing defaults for the character or range of characters specified. When **IMPLICIT UNDEFINED** is specified, the data types of all symbolic names in the program unit that start with a character specified as **IMPLICIT UNDEFINED** must be explicitly declared. For each symbolic name starting with a character specified as **IMPLICIT UNDEFINED**, the compiler issues a diagnostic message if that symbolic name is used but does not appear in an explicit type statement.

INCLUDE Compiler Directive

- **INCLUDE** compiler directive.

INQUIRE Statement

- If `fn` is specified in the **INQUIRE** statement, it must be a valid AIX file name.

NAMELIST Statement

- **NAMELIST** statement.

OPEN Statement

- If `fn` is specified in the **OPEN** statement, it must be a valid AIX file name. If `fn` is omitted, the file name of the file connected to the unit defaults to **`fort.`**`n`, where `n` is the unit specified (`un`), with any leading zeros removed.

PARAMETER Statement

- Names of constants used in the **PARAMETER** statement can be used as part of a **COMPLEX** constant.
- If the name in a **PARAMETER** statement is an integer type of length 1, the constant value is an integer constant that occupies 1 byte of storage. If the name in a **PARAMETER** statement is of logical type of length 2, the constant value is a logical constant that occupies 2 byte of storage.
- The constant in a **PARAMETER** statement can be a hexadecimal constant, an octal constant, a binary constant, or Hollerith constant.
- The constant expression in a **PARAMETER** statement can contain an intrinsic function with constant arguments.

READ Statement

- The character constant, as a format identifier in the **READ** statement, can also be delimited by double quotation marks. A double quotation mark in a constant enclosed in double quotation marks is represented by two consecutive double quotation marks.
- The format identifier on a **READ** statement can be an array name of type double complex.
- If the format identifier of a **READ** statement starts with a `B` or `O` format code, the internal record number is increased by one but no data is transferred.
- Under AIX, all units are initially preconnected for sequential access.
- If the input/output list is not specified, and the format identifier starts with an `A`, `I`, `E`, `F`, `D`, `G`, `L`, `Q`, `Z`, `B`, or `O` format code for the **READ** statement or is empty, a record is skipped over.
- The **NML** option on the **READ** statement (**NAMELIST** with external devices).

- The **NML** option on the **READ** statement (**NAMELIST** with internal files).

RETURN Statement

- The optional expression in a **RETURN** statement can be any arithmetic expression whose value is converted to integer.
- Entities specified as **STATIC** do not become undefined on invocation of a **RETURN** statement.
- Items declared **AUTOMATIC** become undefined.

Statement Function Statement

- If the **IMPLICIT NONE** statement is used within the program unit, the type of the statement function must be explicitly declared in an explicit type statement.

STOP Statement

- If a string of digits is specified in a **STOP** statement, the return code is set to `MOD(n,256)` (because the AIX return code is only one byte).

SUBROUTINE Statement

- Because recursion is allowed, the subroutine name specified in a **SUBROUTINE** statement can appear in a **CALL** statement within the subroutine subprogram. The subroutine name cannot appear as an actual argument in a **CALL** statement or function reference, because it cannot appear in an **EXTERNAL** statement.

WRITE Statement

- The character constant as a format identifier in the **WRITE** statement can be delimited by double quotation marks. A double quotation mark in a constant enclosed in double quotation marks is represented by two consecutive double quotation marks.
- The format identifier on a **WRITE** statement can be an array name of type double complex.
- If the list is not specified in a **WRITE** statement, and the format specification starts with an `A`, `I`, `E`, `F`, `D`, `G`, `L`, `Q`, `Z`, `B`, or `O` or is empty, a blank record is written out.
- If the list is not specified in a **WRITE** statement, and the format specification starts with an `A`, `I`, `E`, `F`, `D`, `G`, `L`, `Q`, `Z`, `B`, or `O` or is empty, the record is filled with blank characters and the relative record number is increased by one.
- The following IBM extension is not available with XL FORTRAN:
  - The **WRITE** statement can be used to write over an end–of–file and extends the external file. An **ENDFILE**, **BACKSPACE**, **CLOSE**, or **REWIND** statement will then reinstate the end–of–file.
- If the length of the record specified in the **RECL** parameter of the **OPEN** statement is smaller than the total amount of data transmitted from the items of the list, as much data as can fit in the record is written, the internal record number is increased by one, and the remainder of the data is ignored.
- The **NML** option on the **WRITE** statement (**NAMELIST** with external devices).
- The **NML** option on the **WRITE** statement (**NAMELIST** with internal files).

- The **NAMELIST** names are folded to lowercase when they are written to the internal file, even if the **MIXED** compiler option is specified.

Intrinsic Functions

- No Q (extended precision) intrinsic functions are provided. If a Q function is specified in a source program, the corresponding D (double precision) function will be used.

Argument List Keywords

- **%VAL**
- **%REF**.

Symbols

A

B

C

D

E

F

G

H

I

K

L

Q

R

S

T

U

V

W

X

Z

Reader's Comment Form

Language Reference for IBM AIX XL FORTRAN Compiler/6000

SC09–1258–00

Please use this form only to identify publication errors or to request changes in publications. Your comments assist us in improving our publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page	Comments

Please contact your IBM representative or your IBM–approved remarketer to request additional publications.

Please print

Date _____

Your Name _____

Company Name _____

Mailing Address _____

_____

_____

Phone No. ( ) _____
Area Code

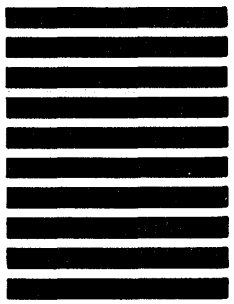
No postage necessary if mailed in the U.S.A

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE



International Business Machines Corporation
Department 997, Building 997
11400 Burnet Rd.
Austin, Texas 78758–3493

Fold

Fold

Cut or Fold Along Line

Fold and Tape

Please Do Not Staple

Fold and Tape

International Business Machines
Corporation
11400 Burnet Road
Austin, Texas 78758–3493

Printed in the
United States of America

SC09-1258-00

SC09-1258-00